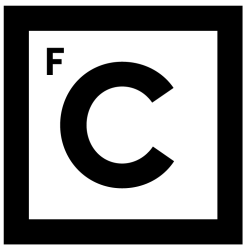
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

Detection of Vulnerabilities and Automatic Protection for Web Applications

Documento provisório

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE CIÊNCIAS DA COMPUTAÇÃO

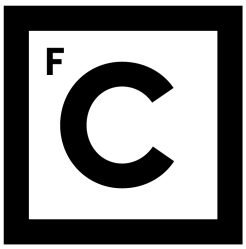
Ibéria Vitória de Sousa Medeiros

Tese orientada pelo Prof. Doutor Miguel Nuno Dias Alves Pupo Correia
e pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Documento especialmente elaborado para a obtenção do grau de doutor

2016

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

Detection of Vulnerabilities and Automatic Protection for Web Applications

Documento provisório

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE CIÊNCIAS DA COMPUTAÇÃO

Ibéria Vitória de Sousa Medeiros

Tese orientada pelo Prof. Doutor Miguel Nuno Dias Alves Pupo Correia
e pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Documento especialmente elaborado para a obtenção do grau de doutor

2016

Resumo

Em duas décadas de existência, a web evoluiu de uma plataforma para aceder a conteúdos hipermédia para uma infraestrutura para execução de aplicações complexas. Estas aplicações têm várias formas, desde aplicações pequenas e caseiras, a aplicações complexas e de grande escala e para diversos propósitos, como por exemplo serviços comerciais como o Gmail, Office 365 e Facebook. Apesar do grande esforço de investigação da última década em como tornar as aplicações web seguras, estas continuam a ser uma fonte de problemas e a sua segurança um desafio. Uma parte importante deste problema deriva de código fonte vulnerável, muitas vezes desenvolvido com linguagens de programação com poucas validações e construído por pessoas sem os conhecimentos mais adequados para uma programação segura. Atualmente a categoria de vulnerabilidades mais explorada é a de validação de *input*, diretamente relacionada com os dados (*inputs*) que os utilizadores inserem nas aplicações web.

A tese propõe metodologias para a detecção e remoção de vulnerabilidades no código fonte e para a proteção das aplicações web em tempo de execução, empregando técnicas como a análise estática de código, aprendizagem máquina e protecção em tempo de execução.

Numa primeira fase, a análise estática é utilizada para descobrir e identificar vulnerabilidades no código programado na linguagem PHP. Os *inputs* dos utilizadores são rastreados e é verificado se estes são parâmetros de funções PHP susceptíveis de serem exploradas. A combinação desta técnica com a aprendizagem máquina aplicada em minerização de dados é proposta para prever se as vulnerabilidades detectadas são falsos positivos ou reais. Caso sejam reais, o resultado da análise estática de código é utilizado para eliminá-las, corrigindo o código fonte automaticamente com *fixes* (remendos) e protegendo assim as aplicações web.

A tese apresenta também uma nova técnica de análise estática de código para descobrir vulnerabilidades. A técnica aprende o que é código vulnerável e depois tira partido desse conhecimento para localizar problemas. A aprendizagem máquina aplicada ao processamento de linguagem natural é utilizada para, numa primeira instância, aprender aspectos que caracterizam as vulnerabilidades, para depois processar e analisar o código fonte, classificando-o como sendo ou não vulnerável, descobrindo e identificando os erros.

Numa terceira fase, é proposta uma nova técnica de proteção em tempo de execução para descobrir e bloquear ataques de injeção contra bases de dados. A técnica é concretizada dentro do sistema de gestão de bases de dados para melhorar a eficácia na detecção dos ataques. É utilizada conjuntamente com identificadores de código fonte que, quando um ataque é sinalizado, permitem identificar a vulnerabilidade no programa.

No total este trabalho permitiu a identificação de cerca de 1200 vulnerabilidades em aplicações web de código aberto disponíveis na Internet, das quais 560 eram até então desconhecidas. As vulnerabilidades desconhecidas foram reportadas aos autores do software onde foram encontradas e muitas delas já foram removidas.

Palavras Chave: aplicações web, segurança de software, vulnerabilidades de validação de input, falsos positivos, análise do código fonte, protecção automática, aprendizagem máquina

Abstract

In less than three decades of existence, the Web evolved from a platform for accessing hypermedia to a framework for running complex web applications. These applications appear in many forms, from small home-made to large-scale commercial services such as Gmail, Office 365, and Facebook. Although a significant research effort on web application security has been on going for a while, these applications have been a major source of problems and their security continues to be challenged. An important part of the problem derives from vulnerable source code, often written in unsafe languages like PHP, and programmed by people without the appropriate knowledge about secure coding, who leave flaws in the applications. Nowadays the most exploited vulnerability category is the input validation, which is directly related with the user inputs inserted in web application forms.

The thesis proposes methodologies and tools for the detection of input validation vulnerabilities in source code and for the protection of web applications written in PHP, using source code static analysis, machine learning and runtime protection techniques.

An approach based on source code static analysis is used to identify vulnerabilities in applications programmed with PHP. The user inputs are tracked with taint analysis to determine if they reach a PHP function susceptible to be exploited. Then, machine learning is applied to determine if the identified flaws are actually vulnerabilities. In the affirmative case, the results of static analysis are used to remove the flaws, correcting the source code automatically thus protecting the web application.

A new technique for source code static analysis is suggested to automatically learn about vulnerabilities and then to detect them. Machine learning applied to natural language processing is used to, in a first instance, learn characteristics

about flaws in the source code, classifying it as being vulnerable or not, and then discovering and identifying the vulnerabilities.

A runtime protection technique is also proposed to flag and block injection attacks against databases. The technique is implemented inside the database management system to improve the effectiveness of the detection of attacks, avoiding a semantic mismatch. Source code identifiers are employed so that, when an attack is flagged, the vulnerability is localized in the source code.

Overall this work allowed the identification of about 1200 vulnerabilities in open source web applications available in the Internet, 560 of which previously unknown. The unknown vulnerabilities were reported to the corresponding software developers and most of them have already been removed.

Keywords: input validation vulnerabilities, web applications, software security, source code static analysis, machine learning, automatic protection.

Resumo Estendido

Desde o seu aparecimento no início dos anos 90, a World Wide Web evoluiu de uma plataforma de acesso a texto e outros elementos multimédia estáticos para a execução de *aplicações web*. Estas aplicações apresentam-se em diversas formas, desde simples aplicações até serviços comerciais de grande escala (ex., Google Docs, Twitter, Facebook), tendo-se gradualmente tornado parte da nossa vida diária. No entanto, as aplicações web têm sido afetadas por vários problemas de segurança com impacto nas organizações. Por exemplo, relatórios recentes mostram um aumento de 33% dos ataques web em 2012, de 62% dos roubos de dados em 2013, de 4% de websites críticos contendo vulnerabilidades em 2014 (Symantec, 2013, 2014, 2015). Sem dúvida que uma razão para a insegurança das aplicações web é que muitos programadores não possuem um conhecimento adequado sobre a construção de código seguro, deixando, portanto, as aplicações com vulnerabilidades.

Embora a segurança tenha começado a ser tomada em consideração durante o desenvolvimento destas aplicações, a tendência para o código fonte conter vulnerabilidades persiste. O OWASP top 10 de 2013 reporta a injecção de SQL e o *cross-site scripting* (XSS) como as duas classes de vulnerabilidades de maior risco (Williams & Wichers, 2013). Embora existam ferramentas para lidar com estas vulnerabilidades, a verdade é que as boas práticas de programação continuam a não ser suficientemente adoptadas e os ataques que exploram tais vulnerabilidades são muito comuns. Tanto a injecção de SQL como o XSS estão incluídas no que denominamos por *vulnerabilidades de validação de input*. Estas são caracterizadas por permitirem que *inputs* maliciosos atinjam certas chamadas a funções, sem terem sido devidamente sanitizados ou validados. Agravando a complexidade das soluções actuais, novas tecnologias estão a tornar-se comuns nas aplicações web. Um exemplo são as base de dados NoSQL, particularmente convenientes para armazenar *big data*. Com as novas tecnologias, surgem também novos vectores de ataque com variadas consequências, como por exemplo,

os 600 TB de dados recentemente roubados do MongoDB (The Hacker News, 2015) (o sistema gestor de bases de dados NoSQL mais utilizado (DB-Engines, 2015)).

A *análise estática de código* é uma das técnicas utilizada pelas empresas para diminuir o problema de vulnerabilidades de software (WhiteHat Security, 2015). As ferramentas de análise estática procuram vulnerabilidades no código fonte, ajudando os programadores a melhorar o código. Esta técnica é eficaz, encontrando potenciais erros nos programas, mas tende a reportar muitos *falsos positivos* (falsas vulnerabilidades) por várias razões, nomeadamente devido à indecidibilidade do problema a resolver (Landi, 1992). Este problema é particularmente difícil de contornar e advém de linguagens de programação como o PHP, as quais são fracamente tipificadas e não formalmente especificadas (Biggar & Gregg, 2009; Biggar *et al.*, 2009). A *análise dinâmica* é uma técnica para encontrar vulnerabilidades em tempo de execução, rastreando os *inputs* dos utilizadores e verificando se eles constituem um ataque (Huang *et al.*, 2003). Os *varredores de aplicações web* utilizam assinaturas para detectar se existem vulnerabilidades específicas numa aplicação, mas, no entanto, esta abordagem tem uma elevada taxa de falsos negativos (não encontra vulnerabilidades existentes) (Vieira *et al.*, 2009). As ferramentas de *fuzzing* e *injecção de ataques* também procuram por vulnerabilidades, mas através da injecção de *inputs* maliciosos (Antunes *et al.*, 2010; Banabic & Candea, 2012). Ao contrário destas técnicas, os mecanismos de *protecção em tempo de execução* não procuram por vulnerabilidades em software, mas detectam ataques que tentam explorá-las (Bandhakavi *et al.*, 2007; Boyd & Keromytis, 2004; Halfond & Orso, 2005; Son *et al.*, 2013).

A *aprendizagem máquina* é uma técnica muito diferente e com um grande leque de aplicações. Nesta tese ela é usada na identificação de vulnerabilidades em código fonte. De facto, as técnicas anteriores que procuram vulnerabilidades e a aprendizagem máquina são, em certo sentido, abordagens disjuntas: os humanos codificam o conhecimento sobre vulnerabilidades versus obtenção automática

deste conhecimento através da aprendizagem máquina. Curiosamente esta dicotomia tem estado presente há muito noutra área da segurança, a detecção de intrusões. Como o seu nome sugere, a detecção de intrusões baseada em comportamento assenta em modelos de comportamento normal criados utilizando técnicas de aprendizagem máquina.

Esta tese enquadra-se no contexto de segurança de software. O objectivo desta tese está relacionado com a investigação de técnicas para a detecção de vulnerabilidades e para a proteção automática de aplicações web. A investigação recai em dois focos principais: *detecção* e *protecção*.

O foco da detecção prende-se com a descoberta e identificação de vulnerabilidades de validação de *input* no código fonte das aplicações web, utilizando para tal a análise estática de código e a aprendizagem máquina. A combinação das duas soluções aplicada em mineração de dados é proposta para prever se as vulnerabilidades detectadas pela primeira são falsos positivos ou reais. Caso sejam reais, o resultado da análise estática de código é utilizado para removê-las, corrigindo o código fonte automaticamente e protegendo assim as aplicações web. A aprendizagem máquina utilizada para descobrir vulnerabilidades também foi experimentada, sendo inovadora nesta área de investigação. A técnica consiste em aprender sobre vulnerabilidades para depois detectá-las. Assim sendo, a aprendizagem máquina aplicada ao processamento de linguagem natural (NLP) é utilizada para, numa primeira instância, aprender as características das vulnerabilidades em código fonte, para depois poder determinar se as aplicações são ou não vulneráveis.

O foco de protecção automática de aplicações web prende remover este tipo de vulnerabilidades pela correção do código fonte e em sinalizar e bloquear ataques de injecção em tempo de execução. A remoção, tal como referido acima, é efectuada utilizando os resultados da análise estática de código. Para bloquear ataques de injeção contra bases de dados utiliza-se uma protecção implementada dentro do sistema gestor de bases de dados (SGBD). Esta opção, para além de melhorar a eficácia na deteção dos ataques, lida também com o problema de *semantic mismatch* existente entre as linguagens de programação do lado do

servidor e o SGBD, quando este interpreta os pedidos (*queries*) compostos e enviados pelas aplicações web. Como forma de localização de vulnerabilidades no código das aplicações, a protecção em tempo de execução é combinada com identificadores de código permitindo a sua descoberta quando um ataque é detectado.

Este trabalho permitiu a identificação de cerca de 1200 vulnerabilidades em aplicações web de código aberto disponíveis na Internet, das quais 560 eram até então desconhecidas. As vulnerabilidades desconhecidas foram reportadas aos autores do software onde foram encontradas e muitas delas já foram removidas.

Acknowledgements

I usually say that there are no first or second places to thank who helps in the realization of a project. This thesis is not an exception, and I hope I do not forget anyone.

To my advisors, Professor Miguel Correia and Professor Nuno Neves, that contributed in different periods and ways. Professor Miguel Correia that is my advisor since my master degree, who taught me everything that I know about research, guiding me with rigor, scientific wisdom, and enthusiasm. I also appreciate his professionalism, time, support, wise advises, motivation and patience, and for believing that this thesis was possible in the form that it was realized, remotely. Professor Nuno Neves that in different times had a great impact in my evolution as researcher. First, transmitting me what was missing in my research, looking for details and teaching me that the little things (even the little ones) are fundamental and make the difference. Secondly, teaching me how to see the opportunities of new research topics and to not waste them. Also, in this last period of the writing of this thesis, the continued concern and friendly support, that I appreciated very much. Thank you, Nuno! Despite not working near them, I had the luck and privilege of having the two advisors I wanted since my master degree.

I would like to thank Professor Paulo Veríssimo who, with his way of teaching security, awoke the "security bug" inside me, and in a certain way put me in this path.

I also want to thank Dr. Miguel Beatriz for his work and dedication, contributing on one of the chapters of this thesis. Professor Armando Mendes from the Department of Mathematics (DM), of University of Azores (Uac), and Professor Bruno Martins from Institute Superior Técnico of University of Lisbon, for their

valuable comments on revisions of papers, and Dra. Alexandra Baptista for the development of the WAP tool logo.

To my friend, life partner, and husband, Pedro Marques, for his support and comprehension, especially in the hard moments, in which he had to abdicate of my company. Also, to Cocas (Inês Gonçalves), my niece, that patiently and curiously followed this thesis, and Maria de Jesus Sousa, mother of my husband, for her support.

I thank my basketball team, that since my teen age showed constant friendship, for all the games and laughs, helping me in hard times, with more stress. In particular thank you Ana Cunha, Fátima Soares Sousa (Fêma), Mónica Campos Cardoso (Mokna), Patricia Índio (Índia Colibri), Patricia Martins (Pat), and Cidália Botelho and Pilar Melo that joined us later.

I express my gratitude to all my students for their comprehension, and Professor Isaura Ribeiro, Professor Jerónimo Nunes, and Professor Maria do Carmo Martins (Miká) from DM, Uac, for their friendship, along these years of doctorate. As well as, to Professor Dulce Domingos, Professor Ana Respício, and the master students of Security Informatics from the Department of Informatics (DI), of Faculty of Sciences, University of Lisbon (FCUL), for their companionship and friendship along the ParIS – ISP event.

Not repeating, but reinforcing, a very special thanks, and expressing my sincere gratitude, to my advisor Miguel and my husband Pedro. This thesis would not exist without Miguel and would not be possible without them.

Finally, I gratefully acknowledge the financial support from European Comission (EC) through projects FP7-607109 (SEGRID) and FP7-257475 (MASSIF), and from national funds through Fundação para a Ciência e a Tecnologia (FCT) with the project RC-Clouds (PTDC/EIA-EIA/115211/2009) and references UID /CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LaSIGE).

Àqueles para quem a aprendizagem é uma fonte de inspiração e paixão.

&

À vida.

Contents

List of Figures

List of Tables

LIST OF TABLES

Code Listings

List of Notations and Acronyms

acc accuracy of classifier

fn false negative outputted by a classifier

fpp false positive rate of prediction

fp false positive outputted by a classifier

$kappa$ kappa statistic

pd probability of detection

pfd probability of false detection

prd precision of detection

$prfp$ precision of prediction

pr precision of classifier

tn true negative outputted by a classifier

tpp true positive rate of prediction

tp true positive outputted by a classifier

$wilcoxon$ Wilcoxon signed-rank test

AI Artificial Intelligence

ASLR Address Space Layout Randomization

AST Abstract Syntax Tree

CMS Content Management System

CS Comment Spamming Injection

DBMS Database Management System

DEKANT hidDEn marKov model diAgNosing vulnerabiliTies

DEP Data Execution Prevention

DoS Denial of Service

DT/PT Directory Traversal or Path Traversal

EI Email Injection

FN False Negatives

FP False Positives

HI Header Injection

HMM Hidden Markov Model

ISL Intermediate Slice Language

K-NN K-Nearest Neighbor

LDAPI LDAP Injection

LFI Local File Inclusion

LR Logistic Regression

ML Machine Learning

MLP Multi-Layer Perceptron

NB Naive Bayes

NLP Natural Language Processing

NoSQLI NoSQL Injection

OSCI OS Command Injection

PHPCI PHP Code Injection

PoS Part-of-Speech

RFI Remote File Inclusion

RT Random Tree

SCD Source Code Disclosure

SEPTIC SElf-Protecting daTabases preventIng attaCks

SQLI SQL Injection

SVM Support Vector Machine

TEPT Tainted Execution Path Tree

TST Tainted Symbol Table

UD Untainted Data

Vul Vulnerabilitiy(ies)

WAF Web Application Firewall

WAP Web Application Protection

WAP-TA Web Application Protection Taint Analyzer

XPathI XPath Injection

XSS Cross-site Scripting

List of Publications

International Conferences and Journals

- Ibéria Medeiros, Nuno Neves, Miguel Correia. *DEKANT: A Static Analysis Tool that Learns to Detect Web Application Vulnerabilities*. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Saarbrücken, Germany, 12 pages, July 2016.
- Ibéria Medeiros, Nuno Neves, Miguel Correia. *Equipping WAP with* WEAPONS *to Detect Vulnerabilities*. In Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 8 pages, June-July 2016.
- Ibéria Medeiros, Miguel Beatriz, Nuno Neves, Miguel Correia. *Hacking the DBMS to Prevent Injection Attacks*. In Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY), New Orleans, EUA, 12 pages, March 2016.
- Ibéria Medeiros, Nuno Neves, Miguel Correia. *Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining*. IEEE Transactions on Reliability, Vol. 65, No. 1, pages 54-69, March 2016.
- Ibéria Medeiros, Nuno Neves, Miguel Correia. *Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives*. In Proceedings of the International World Wide Web Conference (WWW), Seoul, Korea, 12 pages, April 2014.
- Ibéria Medeiros, Nuno Neves, Miguel Correia. *Securing Energy Metering Software with Automatic Source Code Correction*. In Proceedings of

the 11th IEEE International Conference on Industrial Informatics (INDIN), Bochum, Germany, 6 pages, July 2013.

Fast Abstracts

- Ibéria Medeiros, Nuno Neves, Miguel Correia. *Web Application Protection with the WAP tool*. In Supplement of the Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Atlanta, Georgia USA, June 2014.

1 Introduction

Since its appearance in the early 1990s, the World Wide Web evolved from a platform to access text and other media to a framework for running complex *web applications*. These applications appear in many forms, from small home-made to large-scale commercial services (e.g., Google Docs, Twitter, Facebook). However, web applications have been plagued with security problems and the state of their security continues to be a concern. For example, recent reports indicate an increase of web attacks of around 33% in 2012, of data breaches of 62% in 2013, and of critical websites with vulnerabilities of 4% in 2014 (Symantec, 2013, 2014, 2015). Arguably, a reason for the insecurity of web applications is that many programmers lack appropriate knowledge about secure coding, so they leave applications with flaws.

Web application vulnerabilities have been a problem for several years. Although security starts to be taken into account during the application development, the tendency for source the code to contain vulnerabilities persists. The OWASP top 10 2010 reports SQL injection and cross-site scripting as the two classes of vulnerabilities with the highest risk (Williams & Wichers, 2010), and a similar conclusion was reached three years later when there was an update of the study (Williams & Wichers, 2013). Although there are tools to deal with these vulnerabilities, the fact is that good programming practices are still not sufficiently adopted and that the attacks against such vulnerabilities are very common. Both SQL injection and cross-site scripting are what we call *input validation vulnerabilities*. They are characterized by allowing malicious input to reach certain function calls without appropriate sanitization

or validation. To make the problem even more difficult, continuously new technologies are constantly being introduced in web applications increasing their complexity. An example are NoSQL databases, particularly convenient to store big data. With new technologies come also new attack vectors with consequences such as the 600 TB of data recently stolen from the most used (The Hacker News, 2015) NoSQL database, MongoDB (DB-Engines, 2015).

Static analysis is one of the techniques used by companies today to mitigate the problem of software vulnerabilities (WhiteHat Security, 2015). Static analysis tools search for vulnerabilities in source code, helping programmers to improve their applications. This technique is effective to find vulnerabilities, but it tends to report many *false positives* (false vulnerabilities) due to various reasons such as the undecidability of the problem being addressed (Landi, 1992). This difficulty is particularly hard with languages such as PHP because they are weakly typed and not formally specified (Biggar & Gregg, 2009; Biggar *et al.*, 2009). *Dynamic analysis* is another technique to find vulnerabilities by tracking the user inputs at runtime and checking if they constitute an attack (Huang *et al.*, 2003). *Web vulnerability scanners* use signatures to detect if specific vulnerabilities exist in an application, but this approach has been shown to lead to high ratios of false negatives (Vieira *et al.*, 2009). *Fuzzing* and *attack injection* tools also search for vulnerabilities by injecting malicious user inputs (Antunes *et al.*, 2010; Banabic & Candea, 2012). On the contrary to these techniques, *runtime protection* mechanisms do not search for software vulnerabilities, but block or flag attacks that attempt to exploit them (Bandhakavi *et al.*, 2007; Boyd & Keromytis, 2004; Halfond & Orso, 2005; Son *et al.*, 2013).

Machine learning is a very different technique with a large range of applications. However, it can also be applied to search for vulnerabilities in source code. In fact, the previous techniques that look for vulnerabilities and machine learning are in a sense disjoint approaches: humans coding the knowledge about vulnerabilities versus automatically obtaining that knowledge with machine learning. Interestingly this dichotomy has been present for long in another area of security, intrusion detection. As its name suggests, knowledge-based intrusion detection relies on knowledge about intrusions coded by humans (signatures). On the contrary, behavior-based detection relies on models of normal behavior created using machine learning techniques.

1.1 Objectives

This thesis is developed in context of software security, i.e., it proposes methodologies and tools that contribute to build secure applications. The objective is related with the investigation of techniques for the detection of web application vulnerabilities and for the protection of these applications. The thesis has therefore two main focus: *detection* and *protection*.

The first main focus is on identifying and locating input validation vulnerabilities by analyzing statically the source code of web applications, using taint analysis with data mining, and sequence models employed in natural language processing (NLP). The second main focus is on removing these vulnerabilities by fixing the source code of the applications and on blocking injection attacks against databases inside database management systems (DBMS) using for that runtime protections. As a complementary way to find vulnerabilities is based on combining the runtime protection with source code identifiers allowing the identification of the flawed code when an attack is flagged.

Next, we present in more detail the main topics of research and the objectives of the thesis.

Static analysis and data mining

Our general approach consists in analyzing the web application source code searching for vulnerabilities, predicting if the vulnerabilities found are real or false positives, and inserting *fixes* that correct the flaws. This approach also aims at keeping the programmer in the loop of the protection of his web application by allowing him to understand where the vulnerabilities were found and how they were removed.

In order to reduce the number of false positives we propose a new hybrid method of analysis to detect vulnerabilities. We complement a form of static analysis – taint analysis – with the use of machine learning applied to data mining to predict the existence of false positives. This approach is a trade-off between two apparently disjoint approaches: humans coding the knowledge about vulnerabilities (for taint analysis) versus automatically obtaining that knowledge (machine learning, then data mining). We make an argument that the combination of the two broad approaches can be effective for vulnerability location.

The insertion of fixes in the source code of web applications allows the removal of vulnerabilities, but these fixes must not compromise the normal behavior of the applications. Besides studying what pieces of code have to be inserted, we also studied the best places

in the program to insert the fixes and how to avoid breaking the correct behavior of the application.

We implemented this approach in a tool called WAP. Currently, the tool is available at (Medeiros, 2014), having more than 7500 downloads (as *sourceforge* shows). The tool has been included in several relevant projects, such as *OWASP WAP - Web Application Protection* project (Medeiros, 2015), the NIST's SAMATE *Source Code Security Analyzers* (NIST, 2016), and is considered by *Open Source Testing* as a security testing tool (opensourcetesting.org, 2015).

Sequence models

We have also researched an alternative approach to analyze source code to discover vulnerabilities, but in a way that is different from traditional static analysis. The second topic involves using a sequence model to learn to characterize vulnerabilities based on a set of annotated source code slices. The model created can then be used as a static analysis tool to discover and identify vulnerabilities in source code.

We explore the hidden Markov model (HMM) to learn from source code annotated as vulnerable and not vulnerable, and then classify source code elements. The model takes into account the order of the code elements inside the source code being analyzed, and the different states that they can take.

Again, as the approach above, the knowledge is obtained automatically through machine learning, but in this case it is used to train sequence models that usually are applied in NLP. However, on the contrary of the previous approach, this one does not involve coding knowledge about vulnerabilities (all knowledge is learned).

We implemented this approach in a tool called DEKANT, which was used to test both web applications and WordPress plugins (WordPress, 2015).

Runtime protection

The third topic consists in preventing injection attacks against the DBMS behind web applications by embedding protections in the DBMS itself. The motivation is twofold. First, the approach of embedding protections in operating systems and applications running on top of them has been effective to protect this software. Second, there is a semantic mismatch

between how SQL queries are believed to be executed by the DBMS and how they are actually executed, leading to subtle vulnerabilities in prevention mechanisms.

This topic is different from the previous two in terms of objective. Here the goal is to block attacks at runtime, whereas the first two aimed to discover vulnerabilities in source code. Nevertheless, in this case it is also possible to identify vulnerabilities in the source code using the information obtained when an attack is detected.

We implemented this framework in a mechanism called SEPTIC inside the MySQL DBMS, and then tested it with several kinds of classes of attacks and compared it with alternative approaches (e.g., web application firewall).

1.2 Summary of Contributions

This section summarizes the most important contributions that resulted from this work.

Static analysis and data mining to detect and fix vulnerabilities

- A novel hybrid method to detect vulnerabilities with less false positives, then to correct them. After an initial step of taint analysis to flag candidate vulnerabilities, our approach uses data mining to predict the existence of false positives. This approach is a trade-off between two apparently disjoint approaches: humans coding the knowledge about vulnerabilities (for taint analysis) versus automatically obtaining that knowledge (with machine learning, for data mining). Given this more precise form of detection, we do automatic code correction by inserting fixes in the source code.

- The WAP (Web Application Protection) tool that implements this approach for web applications written in PHP with several database management systems. For the implementation of the data mining module a study of the machine learning algorithms was made to select the best three algorithm to be used. This module classifies a vulnerability detected (by taint analysis) as being a false positive or a real vulnerability. The tool is written in the Java programming language so it will be portable to any operating system. The evaluation of the tool was done with a large set of open source PHP applications.

- A study about the fixes that correct the source code and remove the vulnerabilities without compromising the behavior of the web applications. This involves understanding which PHP instructions shall be inserted (fixes) and how they eliminate the vulnerabilities, and where they should be inserted in the source code, changing the semantics of the web application but ensuring the reliability of its functioning.

- A modular and extensible version of the WAP tool that allows creating *weapons* (WAP extensions) to detect and correct new vulnerability classes, without requiring modifications to the core of the tool. A study of the configuration for a new data mining component, with a set of attributes and a data set larger. The evaluation of the new version of the tool was done with a set of open source PHP applications and WordPress plugins.

Learning to detect vulnerabilities statically

- A novel static analysis method to detect vulnerabilities that first learns about them and then later detects them. After an initial step of slices extraction, which start in entry points and end in sensitive sinks, our approach translates them to an intermediate slice language (ISL). Then, it uses a sequence model (HMM) to classify these translated slices as being vulnerable or not, using for that annotated code slices. This approach is based on automatically extracting the knowledge to learn to detect vulnerabilities.

- A sequence model and an intermediate language used by the model to detect vulnerabilities taking into consideration the order in which the code elements appear in the slices. A study about PHP functions that sanitize, validate and modify strings, and the states that an entry point can take when it is an argument of these functions. This involves understanding of which PHP functions and arguments shall be contemplated by the intermediate language and classified by the sequence model.

- The DEKANT (hidDEn marKov model diAgNosing vulnerabiliTies) static analysis tool that implements the approach, learning to detect vulnerabilities using annotated code slices, then using this knowledge to find vulnerabilities in source code of web application written in PHP. The tool is programmed in the Java language so it will be portable.

- An experimental evaluation with a large set of open source PHP applications and WordPress plugins, that shows the ability of this tool to detect previously known and zero-day (i.e., new) vulnerabilities.

Runtime protection and vulnerability identification

- A mechanism to be included in the DBMS to detect and block injection attacks. By being placed inside the DBMS, the mechanism is able to mitigate the semantic mismatch problem and to handle sophisticated SQL injection and stored injection attacks.
- A study about different types of injection attacks and types of semantic mismatch, and their forms of detection, both structurally and syntactically.
- A study about query identifiers to be sent by web applications to the DBMS with the goal of making them unique. A mechanism that, when an attack is detected, allows the location of the vulnerability in the source code exposing information in the query identifier, so that the programmer can remove the flaw.
- The SEPTIC (SElf-Protecting daTabases preventIng attaCks) mechanism implemented in the MySQL DBMS to address injection attacks, independently of the server-side language that was used to develop the applications, and to find the vulnerabilities in source code of web applications. The mechanism is written in the C++ programming language and we explain how it could be adapted to other DBMSs, such as mariaDB and PostgreSQL.
- An experimental evaluation with a set of open source PHP applications and PHP synthetic code that shows the ability of this mechanism to block injection attacks. An evaluation with a testbed with several machines to assess the performance overhead of SEPTIC inside the DBMS.

1.3 Structure of the Thesis

This thesis is organized as follows:

Chapter 2 provides the context in which the thesis appears and presents the related work.

Chapter 3 describes an hybrid approach based on static analysis and data mining to automatically detect vulnerabilities and predict if they are false positives, and then correct the source code to remove the flaws.

Chapter 4 presents how to turn the WAP tool to be extensible in order to detect other classes of vulnerabilities and support different classes of programming frameworks (e.g., WordPress). Also, the experimental evaluation of the tool is presented.

Chapter 5 shows a new approach to detect vulnerabilities statically, based on sequence models used in NLP to learn how to find flaws in the code.

Chapter 6 provides a technique and a mechanism to protect web applications in runtime against injection attacks. The mechanism is inserted inside the DBMS, taking advantage of its resources for a more precise detection.

Chapter 7 concludes the thesis and discusses some future work.

2

Context and Related Work

The main problem in web application security lies arguably in the improper validation of user input. Inputs enter an application through *entry points* (e.g., `$_GET` in the PHP language) and exploit a vulnerability by reaching a *sensitive sink* (e.g., `mysql_query`). Most attacks involve mixing normal input with metacharacters or metadata (e.g., ', OR), and therefore applications often can be protected by placing *sanitization functions* or doing *validation* in the paths between entry points and sensitive sinks.

This chapter provides background on the problem at hand, mainly by introducing the necessary concepts and discussing relevant work done in the area. We organized the related work in four main areas of interest. The first section presents the input validation vulnerabilities handled in our work. Then, we discuss the two main approaches of detection of vulnerabilities – static analysis and fuzzing –, with a strong focus in taint analysis, which is the type of static analysis most used in this work. The third section discusses previous work on the use of machine learning to deal with software vulnerabilities, used in data mining and sequence models. Finally, we present techniques for removing vulnerabilities in source code and protecting applications at runtime.

2.1 Input Validation Vulnerabilities in Web Applications

This section presents the fourteen classes of vulnerabilities considered in our work. Table 2.1 presents them (third column) divided in four categories – *query manipulation, client-side*

injection, file and path injection, and command injection (first column). Columns 2 and 4, respectively, present an overview and the section of each category.

For each vulnerability class, we present how it can be exploited and a technique to avoid its exploitation. This is the technique we used to remove these vulnerabilities (see Section 3.4).

Vulnerability category	Overview	Vulnerability class	Section
Query manipulation	Vulnerabilities related with structures that store data, like databases, and where the malicious code manipulates the queries, changing them	SQL injection XPath injection LDAP injection NoSQL injection	2.1.1
Client-side injection	Vulnerabilities associated to malicious code injected by client-side, such as javaScript, and processed by server-side	Cross-site scripting Header injection Email injection Comment spamming	2.1.2
File and path injection	Class of vulnerabilities that manipulate relative paths or files to, respectively, redirect to a different location or access the local system and web application files	Remote file inclusion Local file inclusion Directory/Path traversal Source code disclosure	2.1.3
Command injection	Vulnerabilities exploited by injection of file system commands and PHP instructions	OS command injection PHP code injection	2.1.4

Table 2.1: Vulnerability classes split by vulnerability categories.

2.1.1 Query manipulation

We consider SQL injection (SQLI), XPath injection (XPathI), LDAP injection (LDAPI) and NoSQL injection (NoSQLI) vulnerabilities as belonging to the same category. They are associated to the construction of queries or filters that are executed by some kind of engine, e.g., a database management system (DBMS). SQLI is the best known and exploited vulnerability (Williams & Wichers, 2013). The other three vulnerabilities behave similarly to SQLI, i.e., if a query is constructed with unsanitized user inputs containing malicious characters, then it is possible to modify the behavior of the executed query (OWASP, 2014b; Scambray *et al.*, 2011). Next, each vulnerability is presented.

SQL injection

SQL injection (SQLI) vulnerabilities are caused by the use of string-building functions to create SQL queries. SQLI attacks mix normal characters with metacharacters to alter

the structure of the query and read or write the database in an unexpected way. Listing 2.1 shows PHP code vulnerable to SQLI. This script inserts in a SQL query (line 4) the username and password provided by the user (lines 2, 3). If the user is malicious, he can provide as username `admin' --` , causing the script to execute a query that returns information about the user *admin* without the need to provide a password: `SELECT * FROM users WHERE username='admin' -- ' AND password='foo'` (note that `--` cause the characters to its right to be interpreted as a comment).

```
1 $conn = mysql_connect("localhost", "username", "password");
2 $user = $_POST['user'];
3 $pass = $_POST['password'];
4 $q = "SELECT * FROM users WHERE username='$user' AND password='$pass'";
5 $result = mysql_query($query);
```

Listing 2.1: PHP login script vulnerable to SQLI.

This vulnerability can be removed either by sanitizing the inputs (e.g., preceding with a backslash metacharacters such as the prime) or by using prepared statements. Sanitization depends on the sensitive sink, i.e., on the way in which the input is used. For the MySQL engine, PHP provides the `mysql_real_escape_string` function. The username could be sanitized in line `2:` `$user = mysql_real_escape_string($_POST['user']);` (note that the same should be done in line 3 to protect the password).

XPath injection

XPath injection (XPathI) works similarly to SQLI, but data is injected in XML documents. XML documents are usually used to store application configuration data or application user information such as user credentials, roles, and privileges (Stuttard & Pinto, 2007). Unlike SQL, XPath does not have a comment character, so if a query contains more than one input parameter the injected code has to be sufficient to build a valid query. Listing 2.2 shows PHP code vulnerable to XPathI. This script inserts in a XPath query (line 4) the username and password provided by the user (lines 2, 3). An attacker can provide as username `admin' or 1=1 or 'a'='b`, causing the script to execute a query that returns information about the user *admin* without the need of giving a password: `//addresses[susername/text()='admin' or 1=1 or 'a'='b' and password/text()="]/creditCard/text()`

```
1 $xml = simplexml_load_file("addresses.xml");
2 $user = $_POST['user'];
3 $pass = $_POST['password'];
4 $query = "//addresses[susername/text()='".$user."' and
    password/text()='".$pass."']/creditCard/text()";
5 $result = $xml->xpath($query);
```

Listing 2.2: PHP login script vulnerable to XPathI.

This vulnerability can be prevented by checking if the user input contains the following malicious characters: `( ) = ' [ ] : , * /`. For the above example, the input would be rejected because the prime character is matched.

LDAP injection

LDAP (Lightweight Directory Access Protocol) injection (LDAPI) vulnerabilities are also exploited by providing metacharacters to string-building functions. Their exploitation aims to modify the structure of the filter and retrieve data from a directory (a hierarchically organized data store (Stuttard & Pinto, 2007)) service over the network, in an unexpected way. However, unlike SQL, LDAP does not contain the comment character, meaning that the malicious input has to be inserted in the first parameter of the filter and contain some filter structure that will cause the intended filter structure to be ignored (Alonso *et al.*, 2009).

The PHP code presented in the Listing 2.3 validates an user in a directory using the username and password credentials provided by the user. This script inserts in a filter (line 6) the required credentials (lines 4, 5). If an attacker provides as username `Bob)(&))` and as password `anyWord`, he causes the script to execute a filter that returns information (*userID*, *name*, *mail* and *creditCard*) about the user *Bob* without the need of providing a correct password. The resulting filter: `(&(username=Bob)(&))` does not contain the second parameter (`(password=$pass)`) because it is substituted by `(&)`. A solution to prevent this vulnerability is to validate the user inputs, checking if they match with some of the following characters `( ) ; , * | & =` (Stuttard & Pinto, 2007).

```
1 $ds = ldap_connect("ldap.server.com");
2 $r = ldap_bind($ds);
3 $dn = "ou=Bank foo of city XXX,o=Bank foo,c=PT";
4 $user = $_POST['user'];
5 $pass = $_POST['password'];
6 $filter = "(&(username=$user)(password=$pass))";
7 $fields = array("userID", "name", "mail", "creditCard");
8 $result = ldap_search($ds, $dn, $filter, $fields);
```

Listing 2.3: PHP login script vulnerable to LDAPI.

NoSQL injection

NoSQL is a common designation for non-relational databases used in many large-scale web applications. There are various NoSQL database models and many engines that implement them. MongoDB (MongoDB, 2015) is the most popular engine implementing the document store model (DB-Engines, 2015). Thereby, we opted for studding the NoSQL injection (NoSQLI) vulnerability in PHP web applications that connect to MongoDB. MongoDB executes queries in JSON format, which is well defined, simple to encode/decode and has good native implementations in many programming languages (Ron *et al.*, 2015). Therefore, a PHP application receives user inputs, represents them as an associative array, and then implicitly encodes the array in JSON.

```
1 $conn = new MongoClient("mongodb.server.com");
2 $db = $conn->selectDB('foo');
3 $collection = new MongoCollection($db, 'users');
4 $user = $_POST['user'];
5 $pass = $_POST['password'];
6 $query = array("username" => $user, "password" => $pass);
7 // line 9 does the following codification implicitly:
8 // $query = "{username: '" + $user + "', password: '" + $pass + "'}";
9 $result = $collection->find($query);
```

Listing 2.4: PHP login script vulnerable to NoSQLI.

Listing 2.4 shows PHP code vulnerable to NoSQLI. This script aims to find a user in the MongoDB database after the username and password are provided by the user (lines 4, 5). If the user is malicious he can sent the following payload `user=admin&password[$ne]=1`,

assigning `admin` to the user parameter (line 4) and changing the password parameter to `password[$ne]` and assigning it the value `1` (line 5). This malicious code generates an associative array `array("username" => "admin", "password" => array("$ne" => 1))` (line 6). It is encoded in JSON as `{username: 'admin', password: { $ne: 1 }}` and sent to be executed by MongoBD (line 9). The query returns information about the user *admin* without the need of giving a correct password, since `$ne` is the not equal condition in MongoDB.

Defending against this vulnerability is possible using one of three measures: (1) casting the parameters received from the user to the proper type (Ron *et al.*, 2015); In Listing 2.4 the username would be cast to `string` by changing `$user = (string)$_POST['user'];` (line 4); (2) using the `mysql_real_escape_string` PHP sanitization function to invalidate the same malicious characters as SQLI, such as the prime; (3) validating the user inputs, checking if they match with some of the following characters `< > & ; / { } : ' * "` (OWASP, 2014b).

2.1.2 Client-side injection

This category of vulnerabilities allows an attacker to execute malicious code (e.g., JavaScript) in the victim's browser. Differently from the other attacks we consider, an attack from this category is not against a web application itself, but against its users. We consider in this category four vulnerability classes that are detailed next: cross-site scripting (XSS), header injection (HI), email injection (EI), and comment spamming injection (CS).

Cross-site scripting

Cross-site scripting (XSS) attacks execute malicious code (e.g., JavaScript) in the victim's browser. There are three main classes of XSS attacks depending on how the malicious code is sent to the victim: reflected or non-persistent, stored or persistent, and DOM-based. In our work, we only consider the first two classes.

A script vulnerable to reflected XSS can have a single line, `echo $_GET['user'];`. The attack involves convincing the user to click on a link that accesses the web application, sending it a script that is reflected by the `echo` instruction and executed in the browser.

A stored XSS is characterized by being executed in two steps: the first involves inserting a malicious JavaScript code in the server-side, then later returning it to one or more users in the

second step. Usually this attack is performed using blogs or forums that allow users to submit data, which is then accessed by other users. The first step can be achieved in two ways: an SQL query that inserts the attacker's script in the database (`INSERT`, and `UPDATE` SQL commands); and contents inserted in a file using, for example, the `file_put_contents` PHP function. Then the attacker's script is retrieved from the database by a `SELECT` SQL command or the file system by the `file_get_contents` PHP function and used in a `echo` statement.

These kinds of attacks can be prevented by sanitizing the input (e.g., `htmlentities` PHP function) and/or by encoding the output. The latter technique consists in encoding metacharacters such as $<$ and $>$ in a way that they are interpreted as normal characters, instead of HTML metacharacters.

Header injection

Header injection (HI) allows an attacker to manipulate the HTTP response, breaking the normal response with the $\backslash n$ and $\backslash r$ characters. This allows the attacker to inject malicious code (e.g., JavaScript) in a new header line or even a new HTTP response, performing in the last case an HTTP response splitting. The vulnerability can be avoided by sanitizing these characters (e.g., substituting them by a space) (Scambray *et al.*, 2011).

Email injection

Email injection (EI) is similar to HI, and consists in an attacker injecting the line termination character or the corresponding encoded character (*%0a* and *%0d*) with the aim of manipulating email components (e.g., sender, destination, message). The same protection method as HI is applied to this vulnerability (Scambray *et al.*, 2011).

Comment spamming injection

Comment spamming injection (CS) has the goal of manipulating the ranking of spammers' web sites, making them appear towards the top of search engines' results. Web applications that allow the users to submit contents with hyperlinks are the potential victims of the attack. Attackers inject, for example, comments containing links to their own web site (Imperva, 2014, 2015). This type of attack works as a stored XSS attack, i.e., it is realized in two steps: first the attacker stores the comments with hyperlinks and secondly the search

engine retrieves these comments and accesses the hyperlinks. To avoid CS, applications have to check the content of posts, looking for hyperlinks (URLs).

2.1.3 File and path injection

Another category considers vulnerabilities dealing with the access to files from web applications or file system, and to URL locations different than the web application. The following vulnerability classes belong to this category: remote file inclusion (RFI), directory traversal or path traversal (DT/PT), local file inclusion (LFI) and source code disclosure (SCD).

Remote file inclusion

PHP allows a script to include files, which can be a vulnerability if the file name takes user input. Remote file inclusion (RFI) attacks exploit this kind of vulnerability by forcing the script to include a remote file containing PHP code. For example, an attack might be to send as parameter *country* the URL `http://www.evil.com/hack` against the script below (Listing 2.5), which would cause the execution of `hack.php` in the server.

```
1 $country = $_GET['country'];
2 include($country . '.php');
```

Listing 2.5: PHP script vulnerable to remote file inclusion.

Directory/Path traversal

A directory traversal or path traversal (DT/PT) attack consists in an attacker accessing unexpected files, possibly outside the web site directory. To access these files, the attacker crafts an URL containing path metacharacters such as .. and /. For instance, if `../../../etc/passwd%00` is passed to the script above, the `/etc/passwd` file is included in the web page and sent to the attacker (the null character %00 truncates additional characters, *.php* in this case).

Local file inclusion

Local file inclusion (LFI) differs from RFI by inserting in a script a file from the file system of the web application (not a remote file). LFI includes local files so the attacker needs

to insert PHP code in the server beforehand, e.g., by injecting PHP code that is written into a log file by calling `http://www.victim.com/<?php+phpinfo();+?>` (this file does not exist, so the URL is logged). The attacker can do a LFI attack by calling the script above with input `/var/log/httpd/error_log%00`. This kind of attack has been motivated by a default configuration introduced in PHP 4.2 that disallows remote file inclusion.

Source code disclosure

The objective of source code disclosure (SCD) attacks is to access web application source code and configuration files. The attackers can use these files to find vulnerabilities and other information useful for attacking the site (e.g., misconfigurations, the database schema) or to steal the application source code itself. This vulnerability normally appears in applications that allow downloading files. Similarly to LFI, SCD may involve a DT/PT attack.

Defending against these kinds of attacks is based on disallowing access to file locations and URL provided by the user. PHP does not provide a sanitization function for this purpose.

2.1.4 Command injection

For the last category we consider operating system command injection (OSCI) and PHP code injection (PHPCI) vulnerabilities as being those that can be exploited by injecting, respectively, operating system commands and PHP code.

OS command injection

An operating system command injection (OSCI) attack consists in forcing the application to execute a command defined by the attacker. Consider as an example a script that uses the following instruction to count the words of a file: `$words=shell_exec("/usr/bin/wc" . $_GET['file']);`. The `shell_exec` function allows the execution of system commands in a shell, whereas the command `wc` is the system command that count the words of a file. The attacker can do command injection by inserting a filename and a command separated by a semi-colon, e.g., `paper.txt; cat /etc/passwd`. The resultant instruction `$words = shell_exec("/usr/bin/wc paper.txt; cat /etc/passwd");` executes the `wc` and `cat` commands. The second command shows the content of the file with the

information of all users of the system.

Despite the PHP language containing `escapeshellarg` and `escapeshellcmd` sanitization functions to avoid OSCI attacks, they do not work correctly in some cases, depending of operating system, such as Windows. So it is preferable to do filtering of the problematic characters, i.e., looking for the following characters `! - # & ; ` | * ? ~ < > ( ) [ ] { } $ \ , ' ^ x0A xFF x2a`.

PHP code injection

The *eval* function runs the PHP code that it receives in its string parameter. A PHP code injection (PHPCI) attack consists in an attacker supplying an input that is executed by an `eval` statement. Consider as example the script of Listing 2.6 that uses the `eval` function to concatenate the string "Hello" with the user name supplied by the user at line 3. The attacker can do command injection by inserting a username and a command separated by a semicolon, e.g., `Bob; cat /etc/passwd`.

```
1 $msg = 'Hello';
2 $x = $_GET['username'];
3 eval('$msg = ' . $msg . $x . ';');
4 echo $msg;
```

Listing 2.6: PHP script vulnerable to PHP code injection.

Defending from this attack is not simple, so the use of the `eval` function is discouraged. PHP has no sanitization function to deal with this problem, so the programmer has to verify the presence of dangerous characters as the semi-colon in the input.

2.2 Detection of Vulnerabilities

The following sections present the two techniques most used for the detection of previously unknown vulnerabilities. While the first – *static analysis* – detects vulnerabilities by analyzing the source code of applications, the other – *fuzzing* – detects vulnerabilities while the application is running. We focus mostly in static analysis, specifically in *taint analysis*, because it is the main technique used in our work to find input validation vulnerabilities in web applications.

Although there are other techniques for the detection of vulnerabilities, such as vulnerability scanners, we do not present them here because they do not discover new vulnerabilities and they execute essentially conventional tests to determine if applications suffer from previously known bugs.

2.2.1 Static analysis

Static analysis tools automate the auditing of code, either source, binary, or intermediate. Static analysis aims to search for potential vulnerabilities by analyzing the code of the applications, without executing them (Chess & McGraw, 2004). The first papers in this area were mostly focused on older vulnerabilities such as buffer overflows and, at least in a case, race conditions (Bishop *et al.*, 1996). Later this type of analysis was extended for binary code (Durães & Madeira, 2006).

Static analysis tools are typically used by programmers during the development of software, checking if the code does not have vulnerabilities. However, these tools only search and detect the vulnerabilities they have been programmed to, searching for patterns and using rules for the type of analysis that they implement (presented in next sections). As consequence of this fact, the tools do not detect newly discovered classes of vulnerabilities in source code, possibly leaving the applications with bugs, generating false negatives – *a vulnerability that exists is not reported*. False negatives are worrying because they lead to a false sense of security, especially if the tool does not report any vulnerability. This would not mean that an application does not contain vulnerabilities and is secure, but that the application does not contain vulnerabilities for which it was checked (Chess & McGraw, 2004).

Conversely, static analysis tools tend to generate false positives – *a non-existent vulnerability ends uo being reported*. This tendency is due to two main reasons, namely, the tools do not implement the kind of analysis that permits to do an effective and precise detection, and the complexity of developing these tools that may lead them to produce wrong paths of analysis. The false positives are also a concern, but in the sense of causing a waste of time, since the programmers have to inspect the code searching for non-existent bugs.

Static analysis techniques can be broadly classified in two main classes, namely lexical analysis and semantic analysis (Bush *et al.*, 2000; Chess & McGraw, 2004; Michael & Lavenhar, 2006; Shankar *et al.*, 2001). Next, these techniques are presented, with more emphasis on taint analysis, a form of semantic analysis.

2.2.1.1 Lexical analysis

This is the most basic form of static analysis. The source code is analyzed to search for library functions or system calls that are not considered reliable – *sensitive sinks* –, i.e., meaning that if malicious data, without sanitization or validation, reaches these functions it can exploit some vulnerability. Examples of these functions are `gets` and `strcpy` in C language, which do not verify the array bounds and therefore can be exploited by malicious code that triggers buffer overflow vulnerabilities.

The tools implementing lexical analysis first parse the source code splitting it in tokens, then compare the tokens with sensitive sinks that are stored in a database. This analysis may generate false positives because, for instance, there may exist variables in source code that have a name equal to a name of a function in the database, and then they are interpreted as being sensitive sinks (Michael & Lavenhar, 2006).

Some old tools that perform this type of analysis are ITS4, Flawfinder and RATS for C and C++ (Chess & McGraw, 2004; Michael & Lavenhar, 2006; Viega *et al.*, 2000). Flawfinder also detects format string and race condition vulnerabilities and RATS analyzes applications written in Perl, PHP and Python.

2.2.1.2 Semantic analysis

Semantic analysis verifies semantic aspects in the source code, such as variable declaration and their bounds, loop control variables and data flow (Chess & West, 2007; Michael & Lavenhar, 2006). This analysis encompasses a set of three main techniques: *type checking*, *control flow analysis* and *data flow analysis*.

Type checking

Type checking is associated with checking bounds of variables, depending of their data type. Some programming languages, such as Java, implement type checking, ensuring that the values assigned to a variable do not exceed the limits of the variable data type. However, other languages, such as C and C++, do not implement such checks and can incur in integer vulnerabilities, namely integer overflow, integer underflow, signedness, and truncation, that can be associated with buffers size. These vulnerabilities if exploited generate, for instance, a buffer overflow and a denial of service (DoS) (Michael & Lavenhar, 2006). This analysis may require annotations in the source code, normally using *type qualifiers* that are specials

keywords written as comments for a specific data type and used to verify the variable bounds. The type qualifiers are ignored by the compiler, but tools that implement this technique obtain information about the data type and limits of the variables, and then they can determine if the values assigned to a variable exceed the data type limits. CQual is a tool that implements this approach for C (Foster *et al.*, 1999; Shankar *et al.*, 2001). BONN is another tool that performs verification of string limits avoiding buffer overflow vulnerabilities in programs written in C (Chess & McGraw, 2004; Wagner *et al.*, 2000; Wilander, 2005).

Lint, LCLint and Splint also are from this category and also analyze C code. The last two are based on the first. The Lint tool verifies the consistency of function calls, checking the possible bounds of the variables that are parameters of functions. When a function is called, the data type of the variables is checked with the data type required by the arguments of the function. Splint (Evans & Larochelle, 2002) is an enhanced version of LCLint (Evans *et al.*, 1994), and does the same type checking as Lint. However, the tool can be extended by the programmers to other forms of checking. The programmers can insert annotations either in the source code of the application or functions of the library, specifying pre- and post-conditions for the functions. The vulnerabilities are detected if these conditions are not matched when the tool analyzes the source code.

Control flow analysis

Another type of static analysis technique is control flow analysis that is used to detect inconsistencies (e.g., vulnerabilities) in source code, by simulating the execution of all possible execution paths of instructions of a program. More precisely, a control flow graph is created taking into account the control flow program instructions (e.g., conditional instructions, loops, call functions), and then the analysis is performed traversing that graph, checking if certain rules are met (Chess & West, 2007). For example, this analysis can be used to detected invalid pointer references, improper operations on system resources (such as trying to close a closed file descriptor), or use of uninitialized memory.

To define the control flow graph, the source code is parsed and an *abstract syntax tree* (AST) is built. Then, the tree is traversed for gathering the control flow paths. Finally, the paths are simulated and the vulnerabilities are detected. The analysis may be realized at three levels: (1) *local*, each function is analyzed separately; (2) *module*, interactions between functions within a specific module are analyzed; (3) *global*, the program is analyzed

globally. PREfix is an error detection tool that implements this type of analysis for C and C++ programs (Bush *et al.*, 2000).

Data flow analysis

Data flow analysis aims to do verifications based on how the data (e.g., user inputs) flows through the code of the program. The tools that employ this technique analyze the code following the data paths inside a program to detect security problems. This analysis can be implemented using a control flow graph, as explained in the last section.

The most commonly used data flow analysis technique is *taint analysis*, which marks the data that enters in the program as *tainted*, and detects if it reaches sensitive functions. Taint analysis typically uses two qualifiers – *tainted* and *untainted* – to annotate the source code, denoting which instructions require/return trustworthy/untrustworthy data. If untrustworthy data reaches a parameter that must be trustworthy, the analysis flags a vulnerability. For example, if a program contains a buffer overflow vulnerability, there is a data flow that starts in an entry point and ends in a function that manipulates a buffer and requires trustworthy data (e.g., the `strcpy` sensitive sink). Therefore, it is possible to track entry points and check if they reach some sensitive sink.

CQUAL (Shankar *et al.*, 2001) is a seminal data flow analysis tool of this kind for C code. It uses the two above-mentioned qualifiers to annotate source code: the *untainted* qualifier indicates either that a function returns trustworthy data (e.g., a sanitization function), or that a parameter of a function requires trustworthy data (e.g., `strcpy` and `mysql_query`); the *tainted* qualifier means that a function returns non-trustworthy data (e.g., functions that read user input such as `$_GET`). The tool tracks the user inputs, changing the data flow state between tainted and untainted, and verifies if tainted data reaches function parameters annotated with *untainted*.

Pixy (Jovanovic *et al.*, 2006) uses taint analysis to verify PHP code, but extends it with *alias analysis* that takes into account the existence of aliases, i.e., of two or more variable names that are used to denominate the same variable. The tool detects SQLI and XSS vulnerabilities in PHP code that do not use objects. The tool is one of the first that processes PHP code.

RIPS (Dahse & Holz, 2014) also uses static analysis to detect input validation vulnerabilities in PHP applications. Like Pixy, the tool uses taint analysis to detect vulnerabilities and, in its first versions, did not support PHP's object-oriented features (Dahse & Holz, 2014).

Later, the RIPS authors enhanced the tool to analyze PHP object-oriented code to search statically for PHP object injection (POI) vulnerabilities that can be exploited by property-oriented programming (POP), i.e., the ability of an attacker to modify the properties of an object that is injected with the aim to exploit a POI vulnerability. This ability allows an attacker to perform code reuse attacks without injecting its own malicious code, but reusing and combining existing code fragments (gadgets) to build a malicious code chain to exploit a POI. They propose an approach to detect POP gadget chains to confirm POI vulnerabilities (Dahse *et al.*, 2014).

SAFERPHP uses static analysis to detect certain semantic vulnerabilities in PHP code: denial of service due to infinite loops, and unauthorized operations in databases (Son & Shmatikov, 2011). In relation to denial of service, the tool uses taint analysis to find loops, and then employs symbolic execution analysis to determine if the terminus of the loops can be prevented by attackers. For the second vulnerability, the tool uses inter-procedural analysis to verify all calling contexts in which sensitive database operations may be invoked, and then employs semantic analysis to identify possible security checks, verifying whether they are present in all calling contexts.

phpSAFE (Fonseca & Vieira, 2014; Nunes *et al.*, 2015) does taint analysis to search for vulnerabilities in PHP code. The authors evaluated experimentally the tool with content management system (CMS) plugins, e.g., WordPress plugins. For more precise detection, the tool not only takes into account the sanitization functions from the web application language, but is also configurable to recognize CMS functions handling entry points, sanitization/validation functions and sensitive sinks.

Yamaguchi et al. (Yamaguchi *et al.*, 2014) presented an approach to do more precise static analysis based on a novel data structure to represent source code that was called code property graph (CPG). This structure combines properties of abstract syntax trees, control flow graphs and program dependence graphs, and gives a comprehensive view on code that allows to detect and create templates for vulnerabilities using graph traversals. The graph traversal navigates over the CPG and accesses the code structure, the control flow, and the data dependencies associated with each node, inspecting thus different code properties and detecting templates for vulnerabilities. Their implementation for static analysis of C code managed to find several new vulnerabilities in the Linux kernel.

Yamaguchi et al. (Yamaguchi *et al.*, 2015) propose a method for inferring search patterns for taint-style vulnerabilities in program written in C language. For a given sensitive sink,

the method identifies corresponding source-sink systems, analyzes the data flow in these systems, and constructs search patterns that model the data flow and sanitization in these systems, reflecting thus the characteristics of taint-style vulnerabilities. They combine static analysis and unsupervised machine learning techniques to generate the patterns. The inferred patterns are represented in a CPG that, when traversed, allows retrieving unsanitized data flows which could be associated to taint-style vulnerabilities. The CPG data structure, which was presented in a previous work of the authors (Yamaguchi *et al.*, 2014), was enhanced by extending it to include information about statement precedence and interprocedural analysis.

AutoISES analyses source code inferring security specifications (e.g., sanitization functions) and using them to detect security violations (Tan *et al.*, 2008). The tool infers that a security check function should be used to protect a particular sensitive operation. For this, first the tool does static analysis of the source code verifying which security check functions are frequently used to protect a given sensitive operation, then infers the security check function to the sensitive operation.

Saner (Balzarotti *et al.*, 2008) addresses the detection of vulnerabilities in PHP code using a combination of static and dynamic analysis (executing a program to check properties and find bugs while it is running (Ball, 1999)). First, during static analysis Saner models how string manipulation routines (e.g., sanitization functions) modify the application's inputs. Secondly, it runs the code with malicious inputs to identify flaws in the sanitization.

Zheng et al. (Zheng & Zhang, 2013) presented an approach to detect remote code execution (RCE) vulnerabilities using a path- and context-sensitive interprocedural analysis. RCE attacks require usually the manipulation of string and non-string parts of the client side inputs, so they propose an analysis that handles these parts in a cohesive and efficient manner, and with multiple PHP scripts and requests. The scripts are analyzed searching for these parts, and encoding them as two kinds of constraints. They developed an algorithm that solves these constrains in an iterative and alternative fashion, so exploits can be composed from this solution.

S3 (Trinh *et al.*, 2014) is a symbolic string solver that addresses the detection of vulnerabilities in web applications by analysis of JavaScript. The solver employs an algorithm for a constraint language, which first makes use of symbolic representation handling string variables, then generates constraint of instances from these symbolic expressions. The results are combined with the specifications for attacks. The specifications are a form of assertions that constitute an attack against a particular sink. Therefore, if the constraint solver finds a

solution to a constraint query, then this represents and attack that can reach a sensitive sink and exploit a vulnerability.

2.2.2 Fuzzing

Fuzzing is another technique to detect vulnerabilities. On the contrary of static analysis it does not analyze the application code to detect vulnerabilities, but in runtime verifies if injected data triggers some vulnerability present in the application. Therefore, it is considered a testing technique that discovers faults in software by feeding a program with unexpected inputs and monitoring that program for exceptions (Evron & Rathaus, 2007; Sutton *et al.*, 2007). This technique tends to be simple to apply because it does not require knowledge about the program to test, and its interaction with the program is limited to the program's entry points (Jimenez *et al.*, 2009).

An important aspect of the technique is how the injected data is generated. It can be implemented based on mutation or generation, meaning that, respectively, the fuzzers mutate existing data samples to create data to be injected and tested, or generate new data based on a model of inputs, i.e., using an input grammar.

The fuzzing technique was first presented by Miller et al. (Miller *et al.*, 1990) that described how they fed UNIX program utilities with random inputs and observed that some of them crashed. Later, some fuzzers, such as SPIKE (Bradshaw, 2010a,b), improve this idea by providing to the applications malformed inputs, using a generic data structure to represent different data types and others based on context-free grammars (CFG) (Kaksonen, 2001; Sutton *et al.*, 2007).

Fuzzers can be classified basically in two categories: *blackbox* and *whitebox* (Sutton *et al.*, 2007). A blackbox fuzzer implements the technique described up to this point. As the blackbox approach is mostly independent of the application and does not require setting up the application, it is useful to mimic the behaviors of attackers while testing existing protections (Duchène *et al.*, 2014). Despite blackbox fuzzers being useful, they tend to find only shallow bugs (bugs that are easy to find) and usually have low code coverage (do not exercise all possible values for a given variable), missing many relevant code paths and thus many bugs (Chipounov *et al.*, 2011; Duchène *et al.*, 2014; Godefroid *et al.*, 2008, 2012). KameleonFuzz is a blackbox fuzzer that searches for XSS vulnerabilities in web applications. It generates malicious inputs to exploit XSS, but also reveals the vulnerability using a genetic

algorithm guided by an attack grammar (Duchène *et al.*, 2014). The tool infers the control flow of the application and combines it with taint flow inference, since XSS vulnerabilities can be discovered by taint analysis. The tool is an extension of two previous works (Doupé *et al.*, 2012; Duchène *et al.*, 2013).

Other technique that, in a certain way, is related with the blackbox fuzzing is attack injection (Antunes *et al.*, 2010). The technique is used to discover automatically vulnerabilities in software. A tool that implements this technique intends to mimic the behavior of an attacker, injecting continuously malformed inputs, while monitoring the application. As expected some attacks are rejected (stopped) by the mechanisms of input validation, while others go through and are processed, performing thus a successful attack. The fuzzing can be used in attack injection to generate the malicious data to be injected and perform the attacks. The AJECT tool follows an approach similar to this principle, i.e., mimics the behavior of attackers to discover vulnerabilities in network-connected servers, for then remove them. The tool monitors the server and the responses returned to the clients, looking for unexpected behaviors, which indicate the presence of a vulnerability that was triggered by some attack. After of attack identification, the tool reproduces the anomaly and uses the information of the attack to support the removal of the vulnerability.

Whitebox fuzzers use symbolic execution and constraint solving applied to the source code (Duchène *et al.*, 2014). The principle of functioning of some whitebox fuzzers is, in a first instance, to generate and inject well-formed inputs in the program and perform dynamic symbolic execution, gathering data flow paths and constraints on inputs from conditional branches encountered along the execution. Then, the collected constrains are negated (constraint solving) and new inputs are injected to collect new execution paths. This process is repeated to gather all possible execution paths and checking several properties in runtime, such as those implemented by Purify, Valgrind, or AppVerifier (Evron & Rathaus, 2007; Sutton *et al.*, 2007). This form of whitebox fuzzing is implemented in the SAGE (Godefroid *et al.*, 2008, 2012), KLEE (Cadar *et al.*, 2008) and DART (Godefroid *et al.*, 2005) fuzzers, using symbolic execution to exercise all possible execution paths of the program. However, as symbolic execution is slow and does not scale to large programs, it is hard to discover deep and complex bugs (Cadar *et al.*, 2008; Chipounov *et al.*, 2011). To deal with this difficulty, Dowser is a fuzzer that combines symbolic execution with dynamic taint analysis to find buffer overflow and underflow (underrun) vulnerabilities buried deep in programs (Haller *et al.*, 2013).

2.3 Vulnerabilities and Machine Learning

Machine learning (ML) and data mining are used in several application areas (e.g., computer games and robotics) and are based on a diverse set of techniques. It is not our intention to describe all these techniques. We will restrict ourselves to the techniques that are used in our work, i.e., to ML applied to data mining and sequence models for the extraction of knowledge for classification. The first two sections present the use of ML with data mining and sequence models. Then, in the next two sections, works using ML to detect software problems, especially vulnerabilities, are presented.

2.3.1 Machine learning classifiers and data mining

Machine learning is a discipline of artificial intelligence (AI) that *gives* computers the ability to learn knowledge without programming (coding) it, and then to use the acquired knowledge to take actions/decisions. Computers have to be guided in order to learn before taking actions. They need a data set of examples – *training data set* – from which to extract knowledge, learning from there.

A task is called *classification* if it aims to assign input objects into classes (a set of discrete values). A *classifier* is an automatic procedure that performs classification. A classifier works as a prediction function that collects *features* that describe an input object and predicts the class that the object fits in, which is the output of the classifier. For example, a spam filter classifies e-mail messages into two classes: spam and not-spam (Hladká & Holub, 2015).

For a correct classification, first the developer should define a list of features that characterize an object, exactly and explicitly based on his intuition about their usefulness. A set of *attributes* is used to represent the features, in which an attribute can represent more than one feature. Sometimes, attributes and features have the same meaning, denoting that an attribute represents a feature. Thereby, since each object is characterized by a list of features, the computer starts by extracting the features, which are next mapped to attributes, composing what we call an *attribute vector*.

An attribute vector together with its class value forms an *instance*. The set of all instances forms a training data set. There are two types of attributes, numerical and categorical. Numerical attributes have numerical values (either discrete or continuous), while categorical attributes have discrete, non-numerical values. A special kind of categorical attributes are

binary attributes that have only two possible values: *true* and *false* (Hladká & Holub, 2015; Witten *et al.*, 2011).

Therefore, classification is a form of data analysis that involves extracting models describing data classes. Such models, called classifiers, predict categorical (discrete, unordered) class labels. Data classification is a two-step process: (1) *learning*, where a classification model is constructed; (2) *classification*, where the model is used to predict class labels for given input data. Because the class label of each training instance is known, this type of ML is known as supervised learning (i.e., the learning of the classifier is *supervised* in the sense that it is told the class each training instance belongs to). An alternative type of ML is unsupervised learning, in which the class label of each training attribute vector is not known, and the number or set of classes to be learned may not be known in advance (Han *et al.*, 2011).

Each classifier uses a machine learning algorithm that depends on the learning type (supervised or unsupervised), then it uses the training data set to discover/extract the knowledge adequate to classify correctly the input data. The selection of machine learning algorithm depends on some factors, such as the type of problem to be solved and the data set nature (Chandola *et al.*, 2009). Examples of some supervised algorithms are decision trees and naive Bayes, and unsupervised algorithms are apriori and K-means.

Data mining, also known as knowledge discovery from data (KDD), aims to turn a large collection of data into knowledge that can assist when taking actions/decisions. In other words, data mining is the analysis of (often large) observational data sets to find patterns (knowledge discovery) and to summarize the data in a way that is understandable and useful to the data owner (Hand *et al.*, 2001). Usually the data collection constitutes a database, and each tuple of the database is composed by attributes that describe objects.

Data mining typically deals with data that has been collected for some purpose different from the data mining itself (Hand *et al.*, 2001). This means that before performing data mining, the collection of data typically has to undergo a pre-processing step to clean the noise (irrelevant and dubious data) and to select the attributes relevant for data mining. Therefore, similarly to ML, data mining has a training data set (a database) constituted by attributes representing features (observational data) that describe input objects, then it extracts knowledge from this database and classifies input data. There are various classification methods used in data mining, being ML one of them, in which adequate and necessary knowledge is

extracted from the database to classify input data. Therefore, we can use ML techniques in data mining databases to build classification models and then classify input data.

2.3.2 Sequence models and natural language processing

Natural language processing (NLP) deals with human–machine communication in both written and spoken natural language. This communication can work if computers are able to recognize proper senses of words in sentences or texts (*sequences of observations*). Data in NLP is represented by written and/or spoken *corpora* (sets of data sets of sequences of observations). Once we computerize objects daily used (e.g., email messages or words in sentences), we can retrieve features from them and form their data representations, which means that the objects become data. On the other hand, machine learning uses data for learning, extracting knowledge and classifying new objects (Hladká & Holub, 2015).

Part-of-Speech (PoS) tagging is one of the most important NLP tasks and uses a classification model for sequences of observations. The task is to assign each word (observation) to a grammatical category (e.g., noun, verb, adjective), named tag. The model's parameters are typically inferred using supervised machine learning techniques, leveraging annotated *corpus* – a data set with sequences of observations annotated manually with the values (e.g., tags in case of PoS) to be learned – to extract rules (knowledge) automatically. Then, with this knowledge, other sequences of observations can be processed and classified. NLP has to take into account the *order* of the observations, as the meaning of sentences depends on this order.

Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical generative model that represents a process as a Markov chain with unobserved (hidden) states. It is a dynamic Bayesian network with nodes that represent random variables and edges that represent probabilistic dependencies between these variables (Baum & Petrie, 1966; Jurafsky & Martin, 2008; Smith, 2011). These variables are divided in two sets: observed variables – *observations* – and hidden variables – *states* (e.g., the states in PoS are the tags). The edges are the transition probabilities, i.e., the probabilities of going from one state to another. States are said to emit observations.

A HMM is composed of:

1. a *vocabulary*, a set of words, symbols or tokens that compose the sequence of observations;

2. *states*, a set of states to classify the observations of the sequence;

3. *parameters*, a set of probabilities: (i) the start-state or initial probabilities, which specify the probability of a sequence of observations starting in each state of the model; (ii) the transition probabilities; (iii) and the emission probabilities, which specify the probability of a state emitting a given observation.

Sequence models are models for structured classification, e.g., for the classification of words in a sentence. The concept of sequence comes from a set of structured sequential observations. Therefore, sequence models correspond to a chain structure (Jurafsky & Martin, 2008) (e.g., the sequence of observations of words in a sentence) to be classified. These models use sequential dependencies in the states, meaning that the *i-th* state depends of the *i-1* previously generated states. In a HMM, the states are generated according to a first order Markov process, in which the *i-th* state depends only of the previous state.

In the context of NLP, a HMM is often used to find the sequence of states that best explain a new sequence of observations, given the learned parameters. This is known as the *decoding problem*, which can be solved by the Viterbi decoding algorithm (Viterbi, 1967). This algorithm uses dynamic programming to pick the best hidden state sequence. Despite the Viterbi algorithm using *bigrams* to generate the *i-th* state, it takes into account all previously generated states, although this is not directly visible. In a nutshell, the algorithm iteratively obtains the probability distribution for the *i-th* state based on the probabilities obtained for the *(i-1)-th* state and the learned parameters.

2.3.3 Detecting vulnerabilities using machine learning

Machine learning has been used in some works to measure software quality by collecting attributes that reveal the presence of software defects (Arisholm *et al.*, 2010; Briand *et al.*, 2000; Lessmann *et al.*, 2008). These works were based on code attributes such as numbers of lines of code, code complexity metrics, and object-oriented features. Some papers went one step further by using similar metrics to predict the existence of vulnerabilities in source

code (Neuhaus *et al.*, 2007; Shin *et al.*, 2011; Walden *et al.*, 2009). They used attributes such as past vulnerabilities and function calls (Neuhaus *et al.*, 2007), or code complexity and developer activities (Shin *et al.*, 2011), or combination of code-metric analysis with meta data gathered from code repositories (Perl *et al.*, 2015). These works did not aim to detect bugs and identify their location, but to assess the quality of the software in terms of the prevalence of defects and vulnerabilities.

PhpMinerI and PhpMinerII are two tools that use data mining to assess the presence of vulnerabilities in PHP programs (Shar & Tan, 2012b,c). These tools extract a set of attributes from program slices that end in a sensitive sink but do not necessarily start in a entry point. The tools are first trained with a set of annotated slices, then apply machine learning algorithms to those attributes to assess the presence of vulnerabilities. The data mining process is not really done by the tools, but instead the user has to use the WEKA tool to do it (Witten *et al.*, 2011). More recently, the authors evolved this idea to use also traces or program execution (Shar *et al.*, 2013). Their approach is an evolution of the previous works that aimed to assess the prevalence of vulnerabilities, but obtaining a higher accuracy.

There are a few static analysis tools that use machine learning techniques in contexts other than web applications. Chucky discovers vulnerabilities by identifying missing checks in C source code (Yamaguchi *et al.*, 2013). The tool does taint analysis to find checks between entry points and sensitive sinks, applies text mining to discover the neighbors of these checks, and then builds a model to identify missing checks. Scandariato et al. use text mining to predict vulnerable software components in Android applications (Scandariato *et al.*, 2014). They use text mining techniques to get the terms (words) present in software components (files) and their frequencies, and use a static code analyzer to check if those software components are vulnerable or not. Then, they correlate the term frequencies in vulnerable software components and build a model to predict if a given software component is vulnerable or not.

2.3.4 Related uses of machine learning

There has been other works that resort to machine learning in the context of software security, including some very recent. SuSi uses machine learning to identify sources and sinks in the source code of the Android API (Rasthofer *et al.*, 2014). It is trained with two annotated sets of sources and sinks, categorized with both syntactic and semantic features, to perform

two distinct classifications. The first, after collecting syntactic features related with methods (functions) from the source code, classifies these methods as being sources or sinks. Then, it uses the second data set composed of semantic features to classify the sources and sinks into a category, such as account, bluetooth (for sources), file, and network (for sinks). After trained with these sets, SuSi managed to detect and identify hundreds of sources and sinks in the Android API.

Soska et al. aim to predict whether a website will become malicious in the future, before it is actually compromised (Soska & Christin, 2014). They use machine learning to retrieve features about the web server and about the websites that it hosts. The features extracted about the websites are for instance: the file system structure (e.g., directory names that indicate that the website is developed using a CMS), webpage structure (e.g., if the web page is generated by a CMS template), and keywords (e.g., presence of some HTML tags). Based on the presence of these features, they predict whether a website will be compromised.

pSigene retrieves features from a large collection of SQL injection attack samples to learn how to characterize them. Then it creates signatures to detect these attacks (Howard *et al.*, 2014). It employs a biclustering technique to identify blocks of attack samples with similar features (a bibluster), and then it uses the logistic regression algorithm to generate the signature for each bicluster (i.e., the logistic regression model that classifies a new sample attack for the bicluster). These signatures are used to create the signatures to be included in an IDS to detect SQL injection attacks.

Nunan et al. also retrieve web document- and URL-based features from a large collection of XSS attacks vectors to learn how to characterize attacks and classify new potential XSS vector attacks as malicious (Nunan *et al.*, 2012). In that large collection, they identified a set of features (obfuscation-based, suspicious patterns and HTML/JavaScript schemes) that allow the accurate classification of XSS in web pages. Then, they analyse automatically web pages to detect XSS attacks, using a three steps process: detection and extraction of obfuscated features, decoding of the web pages and features, and classification of web pages using a machine learning algorithm.

Standard classifiers and other common data mining techniques only look for the presence of attributes, without relating them or considering their order. This can originate wrong classification and prediction. In recent years, this aspect has been taken into consideration for improving accuracy. Specifically, HMM sequence models have started to be used in the context of intrusion detection systems (Bhole & Patil, 2014; Khosronejad *et al.*, 2013;

Sultana *et al.*, 2012). Bhole et al. compare the results of HMM with standard classifiers for the detection of anomalies performed by an IDS. They conclude that the HMM performs better than the others.

Sultana et al. improve the traditional HMM technique used in anomaly detection reducing the time of training. They propose to build a model based on extracted frequent common patterns in trace events instead of taking each trace in its own. The traces are routine calls, since they can reflect the presence of faults, unauthorized usage of resources or unusual function calls due to attacks.

Khosronejad et al. also aim to reduce the time of training during the construction of the HMM. They combine the C5.0 standard classifier with HMM. Thus, first their approach collects the features from IDS events, composing a vector of attributes to be classified by C5.0, then that vector plus the classification are the input for the HMM. The main goal is to verify if the result of C5.0 improves the performance of the HMM in the processing of IDS events and detection of anomalies.

2.4 Removing Vulnerabilities and Runtime Protection

Arguably, the best way of avoid vulnerabilities is writing secure software, but not all programmers have the required knowledge and mistakes can always occur. Static analysis can help for instance to detect and identify vulnerabilities in the source code of applications, which can then be removed. However, not all techniques used in software security have the ability of detection and identification. Some techniques only do some kind of detection, as for instance data mining when predicting the existence of vulnerabilities, while others do neither, such as runtime protection.

This section presents two ways of addressing vulnerabilities. Sections 2.4.1 and 2.4.2, respectively, present the works that remove vulnerabilities by changing the source code and block vulnerability exploitation by interrupting the attack progress.

2.4.1 Removing vulnerabilities

Static analysis is essentially a technique to *detect* vulnerabilities by analyzing the source code of applications. However, it also *identifies* vulnerabilities in the code, meaning that it reports the places in the source code where the vulnerabilities were found. This distinction

is important and beneficial for the programmers because by being told the places in the source code where the vulnerabilities exist, they can correct the programs. This removal is normally done manually by the programmers, but they often do not have adequate training on software security and vulnerabilities. Therefore, it would be beneficial to have tools that detect, identify, and remove vulnerabilities automatically.

WebSSARI (Huang *et al.*, 2004) does static analysis and inserts runtime guards during the analysis. The runtime guards are sanitization routines that are inserted in vulnerable sections of the code that use untrusted information. The tool employs type qualifiers to be associated to variables and functions. The type qualifiers are used to define preconditions for all sensitive sinks, postconditions for sanitization functions that generate trusted output from tainted input, and annotations for all entry points. This information is stored in preludes (files) that are used in the analysis of the code, and after that, a guard is inserted for each variable involved in an insecure statement. Unfortunately, no details are available about what the guards effectively are or how they are inserted, as the tool became commercial around 2006 under the designation of CodeSecure. Interestingly, it seems that the commercial tool no longer corrects applications.

Merlo et al. (Merlo *et al.*, 2007) present a tool that does static analysis of PHP source code and SQL queries. It performs dynamic analysis to build syntactic model-based guards of legitimate SQL queries, and protects queries from input that aims to do SQLI by inserting those guards in the source code. The model guards are SQL abstract syntax trees (ASTs) that are collected by instrumenting the PHP code. They are stored and then be matched against queries sent by the applications before the database accesses. In case of an unmatched query, the model guard is inserted in the source code dynamically, replacing the vulnerable statements.

saferXSS (Shar & Tan, 2012a) does static analysis to find XSS vulnerabilities in Java-based web applications. it applies pattern matching techniques for checking which escape mechanism has to be applied to remove the vulnerabilities, and then prevents exploitation by using functions provided by OWASP's ESAPI (OWASP, 2014a). ESAPI is a web application security control library (like a framework) that implements escape mechanisms and wraps user inputs, preventing input values from causing any script execution. The tool has a detection and a removal phase. The detection phase is based on static analysis, more specifically taint analysis, to identify potential XSS vulnerabilities in the application source code. The

removal phase first verifies the context of each user input referenced in the identified vulnerabilities, next finds the code locations where the untrusted user input can be adequately escaped, determines the required escaping mechanisms, and finally removes the vulnerabilities by applying the appropriate ESAPI functions.

2.4.2 Runtime protection

Static analysis tools are known to report false positives (Jackson & Rinard, 2000; Landi, 1992). Moreover, they generate false negatives because they only find the flaws that they were programmed to detect. Runtime protection is another technique used to improve software security. It follows the principle that: since it is difficult to eliminate all vulnerabilities by analyzing the code, then it is acceptable to protect the applications in runtime. This section presents this form of prevention giving a perspective on how it works and then focusing in prevention against SQLI and stored XSS attacks.

Runtime protection often involves a mechanism that monitors applications, protecting them when an attack is observed. It can be implemented inside of the application to be protected or developed as a third-party software. In both cases, the mechanism is programmed to monitor source code properties of the application that are susceptible to be exploited by malicious data provided by an attacker. In the presence of an attack, it is programmed to take measures, such as stopping the application and logging information about the attack.

Some of the earlier mechanisms of this class aimed at protecting applications from buffer overflows. A buffer overflow can, for example, be characterized by an attacker injecting malicious data and manipulating the memory stack in such way that causes a deviation of the program's flow of control. By overflowing the return address of a function with a specially selected address, the program will execute the code controlled by the attacker. The use of *canaries* is one of the mechanism best known to find this type of attack, which works by detecting invalid changes in the return address. This idea was first proposed in StackGuard (Cowan *et al.*, 1998). The canary is a random number that is associated to a function of the application to be monitored. The application is compiled with the canary, so it is pushed in the stack memory just after the return address. Then, StackGuard checks if the canary was changed before the function returns. Later on, Microsoft integrated this mechanism in their products (Howard & LeBlanc, 2003) and some Linux distributions also include a C compiler with the *Stack Smashing Protector* (Etoh & Yoda, 2002; Wagle & Cowan, 2003).

Another two mechanisms were developed to mitigate buffer overflow attacks, namely *Address Space Layout Randomization* (ASLR) and *Data Execution Prevention* (DEP), both adopted by several operating systems (e.g., Windows and Linux (Howard & LeBlanc, 2003; van de Ven, 2005)). ASLR randomizes the memory addresses where the code and data are loaded into memory, preventing the attacker from knowing what memory addresses should be used to compromise the control flow. DEP marks areas of memory either as executable or nonexecutable, forcing the program to crash in case there is a jump to code in nonexecutable memory. This is used for instance to prevent executable code from being run in the stack segment.

Next we present a set of works on runtime protections against SQLI and XSS, thus more related to our work.

AMNESIA (Halfond & Orso, 2005) and CANDID (Bandhakavi *et al.*, 2007) detect SQLI by comparing the structure of a SQL query before and after the inclusion of user inputs (and before the DBMS processes the queries). Both tools use models to represent the queries and do detection. AMNESIA creates models by analyzing the source code of the application and extracting the query structure. Then, AMNESIA instruments the source code with calls to a wrapper that in runtime compares queries with the models, blocking the attacks. CANDID also analyses the source code of the application to find database queries that handle user inputs, then simulates their execution with benign strings to create the models.

Buehrer et al. (Buehrer *et al.*, 2005) present a similar scheme that manages to detect mimicry attacks by enriching the models (parse trees) with comment tokens. SqlCheck (Su & Wassermann, 2006) is another scheme that compares parse trees to detect SQLI attacks. The detection is made by verifying if the syntactic structure of the query is changed by user inputs containing SQL keywords. For that, it parses the queries and verifies if the nodes of the user inputs have more than one leaf.

DIGLOSSIA (Son *et al.*, 2013) is a technique to detect SQLI attacks that was implemented as an extension of the PHP interpreter. The technique first obtains the query models by mapping all query statements' characters to shadow characters except user inputs, and computes shadow values for all string user inputs. Second, for a query execution it computes the query and verifies if the two parsed trees are isomorphic, i.e., verifies if the root nodes from the two parsed trees are equal.

Recently, Masri et al. (Masri & Sleiman, 2015) and Ahuja et al. (Ahuja *et al.*, 2015) presented two works about prevention of SQLI attacks. The first work presents a tool called

SQLPIL that simply transforms SQL queries created as strings into prepared statements, thus preventing SQLI in the source-code. The second presents three new approaches to detect and prevent SQLI attacks based on rewriting queries, encoding queries and adding assertions to the code.

Dynamic taint analysis tracks the flow of user inputs in the application and verifies it they reach dangerous instructions. Xu et al. (Xu *et al.*, 2005) show how this technique can be used to detect SQLI and reflected XSS. They annotate the arguments from source functions and sensitive sinks as untrusted and instrument the source code to track the user inputs to verify if they reach the untrusted arguments of sensitive sinks (e.g., functions that send queries to the database). A different but related idea is implemented by CSSE to protect PHP applications from SQLI, XSS and OSCI. CSSE modifies the platform to distinguish between the parts of a query that come from the program and from the external (input), defining checks to be performed on the latter (Pietraszek & Berghe, 2005). An example check is to verify if the query structure becomes different due to inputs. WASP also does something similar to block SQLI attacks (Halfond *et al.*, 2008).

Valeur et al. present an anomaly-based intrusion detection system for SQLI attacks (Valeur *et al.*, 2005). During the training phase the detector creates a model (a set of profiles) of normal access to the database. In runtime it detects deviations from that model.

The idea of randomized instruction sets was first proposed to block binary code injection attacks (Barrantes *et al.*, 2003; Kc *et al.*, 2003). The RISE mechanism works as a unique and private machine instruction set for each executing program. The code of the program is protected by scrambling each byte with random numbers seeded with a random key that is unique to each program execution. When binary code is injected, it will be descrambled resulting in random bits that probably will crash the program, as the code was not correctly scrambled (Barrantes *et al.*, 2003). A similar mechanism was proposed by Kc et al. (Kc *et al.*, 2003). Boyd et al. evolved this idea for protecting web applications from SQLI and presented the SQLrand tool (Boyd & Keromytis, 2004). SQLrand creates a new SQL language by remapping SQL keywords with a secret key, essentially by appending a number to every SQL keyword. Applications must be modified to use the new language for the generated queries. In runtime, SQLrand decodes the queries to the original SQL keywords, and then sends them to the DBMS. Before decoding, the tool checks if an original SQL keyword appears in the user inputs, detecting SQLI of first and second order that alter the query structure.

There are several tools to detect reflected XSS vulnerabilities and attacks in the literature (Kieyzun, A. et al., 2009; Papagiannis *et al.*, 2011; Saxena *et al.*, 2010). Gálan et al. propose a vulnerability scanner to detect this attack by finding the entry points of the application susceptible to be exploited. Then, they inject malicious data in those entry points while crawling the web application to verify if the injected data is returned (Gálan *et al.*, 2010). Using source code static analysis, Wang et al. identify the slice among the first step of the vulnerability (functions that write in the database) and the second step (sensitive sinks) extract the vulnerable slice to look for the existence of stored XSS (Wang *et al.*, 2011).

3

Detecting and Removing Vulnerabilities with Static Analysis and Data Mining

Arguably, a reason for the insecurity of web applications is that many programmers lack appropriate knowledge about secure coding, so they leave applications with flaws. However, the mechanisms for web application security fall in two extremes. On one hand, there are techniques that put the programmer aside, e.g., web application firewalls and other runtime protections (Halfond *et al.*, 2008; Pietraszek & Berghe, 2005; Wang *et al.*, 2006). On the other hand, there are techniques that discover vulnerabilities but put the burden of removing them on the programmer, e.g., black-box testing (Antunes *et al.*, 2010; Banabic & Candea, 2012; Huang *et al.*, 2003) and static analysis (Huang *et al.*, 2004; Jovanovic *et al.*, 2006; Shankar *et al.*, 2001).

This chapter explores an approach for automatically protecting web applications while keeping the programmer in the loop. The approach consists in analyzing the web application source code searching for input validation vulnerabilities, and inserting fixes in the same code to correct these flaws. The programmer is kept in the loop by being allowed to understand where the vulnerabilities were found, and how they were corrected. This approach contributes directly to the security of web applications by removing vulnerabilities, and indirectly by letting the programmers learn from their mistakes. This last aspect is enabled by inserting fixes following common security coding practices, so programmers can learn these practices by seeing the vulnerabilities and how they were removed.

We explore the use of a novel combination of methods to detect this type of vulnerabilities: static analysis with data mining. Static analysis is an effective mechanism to find vulnerabilities in source code, but tends to report many false positives (non-vulnerabilities) due to its undecidability (Landi, 1992). This problem is particularly difficult with languages such as PHP that are weakly typed and not formally specified (de Poel, 2010). Therefore, we complement a form of static analysis, taint analysis, with the use of data mining to predict the existence of false positives. This solution combines two apparently disjoint approaches: humans coding the knowledge about vulnerabilities (for taint analysis), in combination with automatically obtaining that knowledge (with supervised machine learning supporting data mining).

To predict the existence of false positives, we introduce the novel idea of assessing if the vulnerabilities detected are false positives using data mining. To do this assessment, we measure attributes of the code that we observed to be associated with the presence of false positives, and use a combination of the three top-ranking classifiers to flag every vulnerability as false positive or not. We explore the use of several classifiers: ID3, C4.5/J48, Random Forest, Random Tree, K-NN, Naive Bayes, Bayes Net, MLP, SVM, and Logistic Regression (Witten *et al.*, 2011). Moreover, for every vulnerability classified as false positive, we use an induction rule classifier to show which attributes are associated with it. We explore the JRip, PART, Prism, and Ridor induction rule classifiers for this goal (Witten *et al.*, 2011). Classifiers are automatically configured using machine learning based on labeled vulnerability data.

Ensuring that the code correction is done correctly requires assessing that the vulnerabilities are removed, and that the correct behavior of the application is not modified by the fixes. We propose using program mutation and regression testing to confirm, respectively, that the fixes function as they are programmed to (blocking malicious inputs), and that the application remains working as expected (with benign inputs).

The chapter also describes the design of the *Web Application Protection* (WAP) tool that implements our approach (Medeiros, 2014). WAP analyzes and removes input validation vulnerabilities from programs or scripts written in PHP 5, which according to a recent report is used by more than 82% of existing web applications (Imperva, 2014). WAP covers eight classes of vulnerabilities presented in Section 2.1, namely SQLI, XSS, RFI, LFI, SCD, DT/PT, OSCI and PHPCI. Currently, WAP assumes that the background database is MySQL,

DB2, or PostgreSQL. The tool might be extended with more flaws and databases (see Chapter 4), but this set is enough to demonstrate the concept. Designing and implementing WAP was a challenging task. The tool does taint analysis of PHP programs, a form of data flow analysis. To do a first reduction of the number of false positives, the tool performs global, interprocedural, and context-sensitive analysis, which means that data flows are followed even when they enter new functions and other modules (other files). This result involves the management of several data structures, but also deals with global variables (that in PHP can appear anywhere in the code, simply by preceding the name with *global* or through the $_GLOBALS array), and resolving module names (which can even contain paths taken from environment variables). Handling object orientation with the associated inheritance and polymorphism was also a considerable challenge.

This chapter describes a form to detect and correct automatically the eight classes of vulnerabilities mentioned above, predicting if they are false positives or not. In Section 3.1 the approach to detect and correct automatically this type of vulnerabilities is discussed, using the output of taint analysis and predicting false positives by data mining, and the architecture of the WAP tool that implements the approach is presented. The tool is composed by three main modules - Code Analyzer, False Positives Predictor and Code Corrector - discussed in Sections 3.2, 3.3 and 3.4, respectively. The first module performs taint analysis to detect candidate vulnerabilities, while the second classifies them as being or not false positives, and the third removes the true positives (vulnerabilities) by correction of the source code. Section 3.5 presents the challenges to implement the WAP tool and Section 3.6 shows an experimental evaluation of the tool. The chapter ends with conclusions of this form of detection and correction (Section 3.7), and discusses some related work.

3.1 A Hybrid of Static Analysis and Data Mining

3.1.1 Overview of the approach

The notion of detecting and correcting vulnerabilities in the source code that we propose is tightly related to *information flows*: detecting problematic information flows in the source code, and modifying the source code to block these flows. The notion of information flow

is central to two of the three main security properties: confidentiality and integrity (Sandhu, 1993). Confidentiality is related to private information flowing to public objects, whereas integrity is related to untrusted data flowing to trusted objects. Availability is an exception as it is not directly related to information flow.

The approach proposed is, therefore, about *information-flow security* in the context of web applications. We are mostly concerned with the server-side of these applications, which is normally written in a language such as PHP, Java, or Perl. Therefore, the problem is a case of language-based information-flow security, a topic much investigated in recent years (Huang *et al.*, 2004; Nguyen-Tuong *et al.*, 2005; Sabelfeld & Myers, 2003). Attacks against web vulnerabilities can be expressed in terms of violations of information-flow security. Figure 3.1 shows the information flows that exploit each of the vulnerabilities of Section 2.1. The information flows are labeled with the vulnerabilities that usually permit them (a few rarer cases are not represented). XSS is different from other vulnerabilities because the victim is not the web application itself, but a user. Our approach is a way of enforcing information-flow security at the language-level. The tool detects the possibility of the existence of the information flows represented in the figure, and modifies the source code to prevent them.

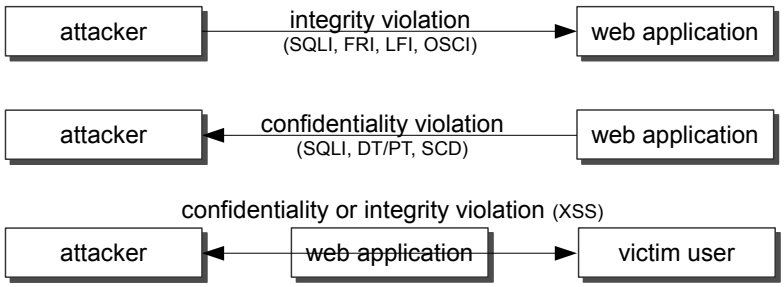


Figure 3.1: Information flows that exploit web vulnerabilities.

The approach can be implemented as a sequence of steps.

1. *Taint analysis*: parsing the source code, generating an abstract syntax tree (AST), doing taint analysis based on the AST, and generating trees describing candidate vulnerable control-flow paths (from an entry point to a sensitive sink).

2. *Data mining*: obtaining attributes from the candidate vulnerable control-flow paths, and using 3 classifiers to predict if each candidate vulnerability is a false positive or

not. In the presence of a false positive, use induction rules to present the relation between the attributes that classified it.

3. *Code correction*: given the control-flow path trees of vulnerabilities (predicted not to be false positives), identifying the vulnerabilities, the fixes to insert, and the places where they have to be inserted; assessing the probabilities of the vulnerabilities being false positives; and modifying the source code with the fixes.

4. *Feedback*: provide feedback to the programmer based on the data collected in the previous steps (vulnerable paths, vulnerabilities, fixes, false positive probability, and the attributes that were used to classify a false positive).

5. *Testing*: higher assurance can be obtained with two forms of testing, specifically program mutation to verify if the fixes do their function, and regression testing to verify if the behavior of the application remains the same with benign inputs.

3.1.2 Architecture

Figure 3.2 shows the architecture that implements steps 1 to 4 of the approach (testing, which is step 4, is not represented). It is composed of three modules: code analyzer, false positives predictor, and code corrector. The *code analyzer* first parses the PHP source code and generates an AST. Then, it uses tree walkers to do *taint analysis*, i.e., to track if data supplied by users through the entry points reaches sensitive sinks without sanitization. While doing this analysis, the code analyzer generates tainted symbol tables and tainted execution path trees for those paths that link entry points to sensitive sinks without proper sanitization. The *false positives predictor* continues where the code analyzer stops. For every sensitive sink that was found to be reached by tainted input, it tracks the path from that sink to the entry point using the tables and trees just mentioned. Along the track paths (slice candidate vulnerabilities in the figure), the vectors of attributes (instances) are collected and classified by the data mining algorithm as true positive (a real vulnerability), or false positive (not a real vulnerability). Note that we use the terms true positive and false positive to express that an alarm raised by the taint analyzer is correct (a real vulnerability) or incorrect (not a real vulnerability). These terms do not mean the true and false positive rates resulting from the data mining algorithm, which measure its precision and accuracy.

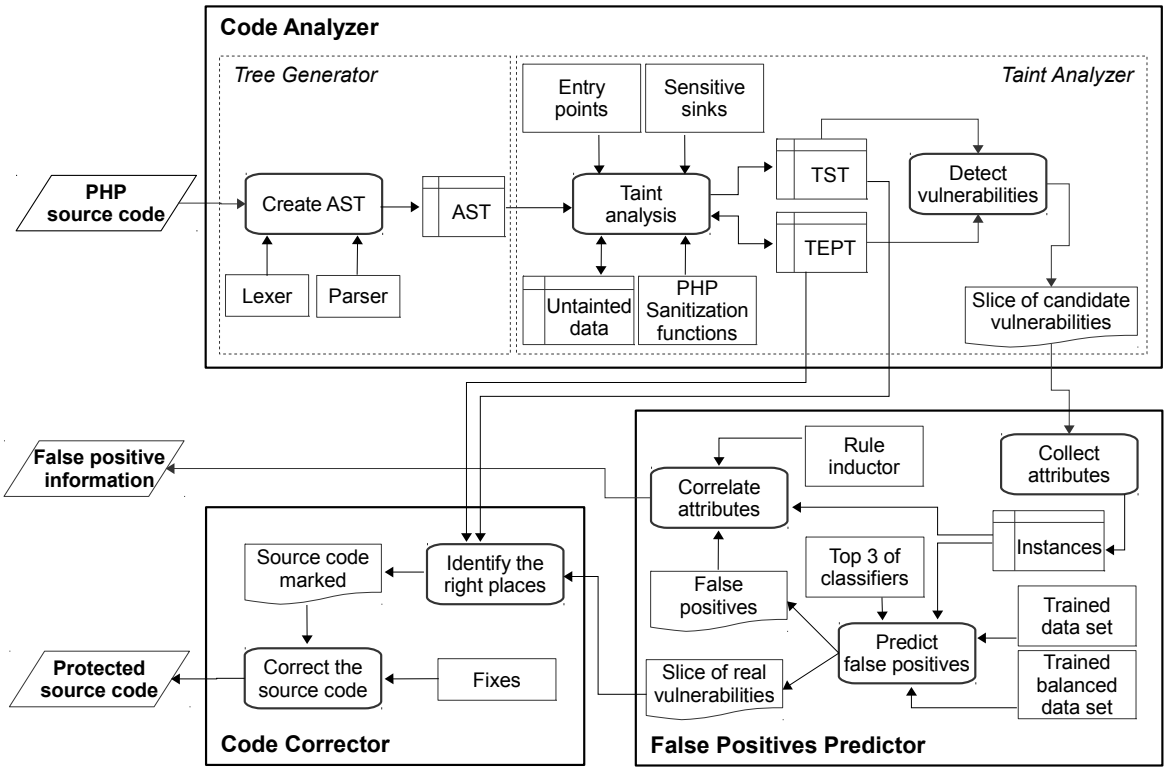


Figure 3.2: Architecture including main modules, and data structures.

The *code corrector* picks the paths classified as true positives to signal the tainted inputs to be sanitized using the tables and trees mentioned above. The source code is corrected by inserting fixes, e.g., calls to sanitization functions. The architecture describes the approach, but represents also the architecture of the WAP tool.

3.2 Detecting Candidate Vulnerabilities by Taint Analysis

Taint analysis for vulnerability detection has been investigated for more than a decade (Evans & Larochelle, 2002). However, papers in the area do not present the process in detail, and usually do not do interprocedural, global, and context-sensitive analysis, so we present how we do it. The taint analyzer is a static analysis tool that operates over an AST created by a lexer and a parser, for PHP 5 in our case (in WAP we implemented it using ANTLR (Parr, 2009)). In the beginning of the analysis, all *symbols* (variables, functions) are *untainted*

unless they are an entry point (e.g., `$a` in `$a = $_GET['u']`). The tree walkers (also implemented using the ANTLR) build a *tainted symbol table* (TST) in which every cell is a program statement from which we want to collect data (see Figure 3.3). Each cell contains a subtree of the AST plus some data. For instance, for statement `$x = $b + $c;` the TST cell contains the subtree of the AST that represents the dependency of `$x` on `$b` and `$c`. For each symbol, several data items are stored, e.g., the symbol name, the line number of the statement, and the taintedness.

Taint analysis involves traveling though the TST. If a variable is tainted, this state is propagated to symbols that depend on it, e.g., function parameters or variables that are updated using it. Figure 3.3 (iii) shows the propagation of the taintedness of the symbol `$_GET['u']` to the symbol `$a`, where the attribute *tainted* of `$a` receives the value of the attribute *tainted* from `$_GET['u']`. On the contrary, the state of a variable is not propagated if it is untainted, or if it is an argument of a PHP sanitization function (a list of such functions is in Section 3.5). The process finishes when all symbols are analyzed this way.

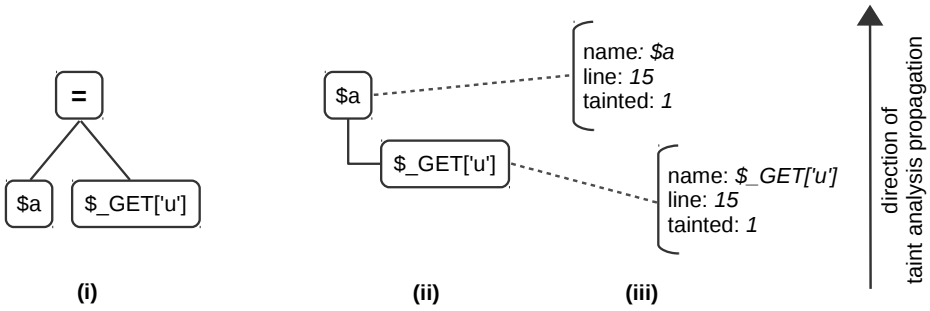


Figure 3.3: Example (i) AST, (ii) TST, and (iii) taint analysis.

While the tree walkers are building the TST, they also build a *tainted execution path tree* (TEPT; example in Figure 3.4 b)). Each branch of the TEPT corresponds to a tainted variable, and contains a sub-branch for each line of code where the variable becomes tainted (a square in the figure). The entries in the sub-branches (curly parentheses in the figure) are the variables that the tainted variable propagated its state into (dependent variables). Taint analysis involves updating the TEPT with the variables that become tainted.

Figure 3.4 shows a sample script vulnerable to SQLI, its TEPT, and *untainted data* (UD) structures. The analysis understands that `$a` and `$b` are tainted because they get non-sanitized values from an entry point (lines 1-2). When analyzing line 3, it finds out that `$c` is

```
1  $a = $_GET['user'];
2  $b = $_POST['pass'];
3  $c = "SELECT * FROM users WHERE u = '".mysql_real_escape_string($a)."'";
4  $b = "wap";
5  $d = "SELECT * FROM users WHERE u = '".$b."'";
6  $r = mysql_query($c);
7  $r = mysql_query($d);
8  $b = $_POST['pass'];
9  $query = "SELECT * FROM users WHERE u = '".$a."' AND p = '".$b."'";
10 $r = mysql_query($query);
```

a) Sample script vulnerable to SQLI.

b) TEPT of a)

c) untainted data of a)

Figure 3.4: Script with SQLI vulnerability, its TEPT, and untaint data structures.

not tainted because `$a` is sanitized. Analyzing line 5, `$d` is not tainted because `$b` becomes untainted in line 4. In line 8, `$b` is tainted again; and in line 9, `$query` becomes tainted due to `$a` and `$b`. A vulnerability is flagged in line 10 because tainted data reaches a sensitive sink (`mysql_query`). When `$a` becomes tainted, a new branch is created (Figure 3.4 b). Also, a sub-branch is created to represent the line of code where `$a` became tainted. The

same procedure occurs to $\$b$ in line 2. The state of `$b` in line 4 becomes untainted. An entry of it is added to UD (Figure 3.4 c) to avoid its taintedeness propagation from TEPT. So, in line 5, the statement is untainted because `$b` belongs to UD, and its taintedness propagation is blocked. When, in line 8, `$b` becomes tainted again, a new sub-branch is created in `$b` to line 8, and its entry is removed from UD. For `$query`, a branch with a sub-branch representing line 9 is created. Here, `$query` is tainted because `$a` and `$b` propagated their taintedness, so an entry of `$query` is added in the last sub-branch created in `$a` and `$b` (1: to `$a`; 8: to `$b`). Analyzing line 10, `mysql_query` and `$r` become tainted because `$query` taintedness is propagated. The procedure is repeated for the creation of the branch and insertion of the dependency in the sub-branch. As we can see, the process of taint analysis is a symbiosis of exploring the TST, TEPT, and UD structures. A symbol from a statement of TST propagates its taintedness to its root node iff it belongs to TEPT but not to UD. At the end of the analysis of a statement, the TEPT or UD or both are updated: TEPT with new tainted variables and tainted dependent variables, and UD with the addition or the removal of variables.

To summarize, the taint analysis model has the following steps.

1. Create the TST by collecting data from the AST, and flagging as tainted the entry points.

2. Propagate taintedness by setting variables as tainted in the TST iff the variable that propagates its taintdeness belongs to the TEPT and not to the UD.

3. Block taintedness propagation by inserting in the UD any tainted variable that belongs to the TEPT and is sanitized in the TST; conversely, remove a variable from the UD if it becomes tainted.

4. Create the TEPT: (i) a new branch is created for each new tainted variable resulting from the TST; (ii) a sub-branch is created for each line of code where the variable becomes tainted; and (iii) an entry in a sub-branch is made with a variable that becomes tainted by the taintedness propagation from the branch variable.

5. Flag a vulnerability whenever a TST cell representing a sensitive sink is reached by a tainted variable in the same conditions as in step 2.

During the analysis, whenever a variable that is passed to a sensitive sink becomes tainted, the false positives predictor is activated to collect the vector of attributes, creating thus an instance, and classify the instance as being a false positive or a real vulnerability. In the last case, the code corrector is triggered to prepare the correction of the code. The code is updated and stored in a file only at the end of the process, when the analysis finishes, and all the corrections that have to be made are known.

Table 3.1 shows the functions used to fix the vulnerabilities detected. For example, for SQLI the tool uses the function `san_sqli` (developed by us) that employs sanitization functions provided by PHP (column on the right hand side of the table), but also replaces some problematic, deprecated, tainted sensitive sinks (`mysql_db_query`, `mysqli_execute`) by non-deprecated functions with similar functionality (`mysql_query`, `mysqli_stmt_execute`).

3.3 Predicting False Positives

The static analysis problem is known to be related to Turing's halting problem, and therefore is undecidable for non-trivial languages (Landi, 1992). In practice, this difficulty is solved by making only a partial analysis of some language constructs, leading static analysis tools to be unsound. In our approach, this problem can appear, for example, with string manipulation operations. For instance, it is unclear what to do to the state of a tainted string that is processed by operations that return a substring or concatenate it with another string. Both operations can untaint the string, but we cannot decide with complete certainty. We opted to let the string be tainted, which may lead to false positives but not false negatives.

The analysis might be further refined by considering, for example, the semantics of string manipulation functions, as in (Wassermann & Su, 2007). However, coding explicitly more knowledge in a static analysis tool is hard, and typically has to be done for each class of vulnerabilities ((Wassermann & Su, 2007) follows this direction, but considers a single class of vulnerabilities, SQLI). Moreover, the humans who code the knowledge have first to obtain it, which can be complex.

Data mining allows a different approach. Humans label samples of code as vulnerable or not, then machine learning techniques are used to configure the tool with knowledge acquired from the labelled samples. Data mining then uses that data to analyze the code. The key idea

Vulnerability	Entry points	Sensitive sinks	Sanitization functions used for untainting	Sanitization functions used for correction [3]
SQL Injection	$_GET $_POST $_COOKIE $_REQUEST HTTP_GET_VARS HTTP_POST_VARS HTTP_COOKIE_VARS HTTP_REQUEST_VARS	**MySQL**		
		mysql_query mysql_unbuffered_query mysql_db_query [1] mysqli_query mysqli_real_query mysqli_master_query mysqli_multi_query mysqli_stmt_execute mysqli_execute [2] mysqli::query mysqli::multi_query mysqli::real_query mysqli_stmt::execute	mysql_escape_string mysql_real_escape_string mysqli_escape_string mysqli_real_escape_string mysqli_stmt_bind_param mysqli::escape_string mysqli::real_escape_string mysqli_stmt::bind_param	***san_sqli***, that uses the following PHP sanitization functions, by DBMS: mysql_real_escape_string mysqli_real_escape_string mysqli_stmt_bind_param mysqli::real_escape_string mysqli_stmt::bind_param
		DB2		
		db2_exec	db2_escape_string	db2_escape_string
		PostgreSQL		
		pg_query pg_send_query	pg_escape_string pg_escape_bytea pg_escape_literal	pg_escape_string pg_escape_bytea pg_escape_literal
Remote File Inclusion Local File Inclusion Directory Traversal/ Path Traversal	$_GET $_POST $_COOKIE $_REQUEST HTTP_GET_VARS HTTP_POST_VARS HTTP_COOKIE_VARS HTTP_REQUEST_VARS	fopen, copy, unlink file_get_contents, file require, require_once include, include_once move_uploaded_file imagecreatefromgd2 imagecreatefromgd2part imagecreatefromgd imagecreatefromgif imagecreatefromjpeg imagecreatefrompng imagecreatefromstring imagecreatefromwbmp imagecreatefromxbm imagecreatefromxpm		***san_mix***, that performs validation by black-list
Source Code Disclosure		readfile highlight_file		
OS Command Injection		passthru, system, shell_exec, exec, pcntl_exec, popen, proc_open		***san_osci***, that performs validation by black-list
Cross Site Scripting	$_GET $_POST $_COOKIE $_REQUEST HTTP_GET_VARS HTTP_POST_VARS HTTP_COOKIE_VARS HTTP_REQUEST_VARS $_FILES $_SERVERS	echo, print, printf die, error exit	htmlentities htmlspecialchars strip_tags urlencode	***san_out*** [4]
		file_put_contents, fprintf		***san_wdata*** [4]
		file_get_contents fgets, fgetc, fscanf		***san_rdata*** [4]
PHP Code Injection		eval, preg_replace		***san_eval***, that performs validation by black-list and sanitization

[1] Function deprecated replaced by *mysql_query* function. [2] Function deprecated replaced by *mysqli_stmt_execute* function.
[3] WAP-specific sanitization functions. [4] Uses the OWASP PHP Anti-XSS Library v1.2b.

Table 3.1: Sanitization functions used to fix PHP code by vulnerability and sensitive sink.

is that there are symptoms in the code, e.g., the presence of string manipulation operations, that suggest that flagging a certain pattern as a vulnerability may be a false positive (not a vulnerability). The assessment has mainly two steps, as follows.

1. *definition of the classifier:* pick a representative set of vulnerabilities identified by the taint analyzer, verify if they are false positives or not, extract a set of attributes, analyze their statistical correlation with the presence of a false positive, evaluate candidate classifiers to pick the best for the case in point, and define the parameters of the classifier.

2. *classification of vulnerabilities:* given the classifier, for every vulnerability found determine if it is a false positive or not.

3.3.1 Classification of vulnerabilities

Any process of classification involves two aspects: the *attributes* that allow classifying an instance, and the *classes* in which these instances are classified. We identified the *attributes* by analyzing manually a set of vulnerabilities found by WAP's taint analyzer. We studied these vulnerabilities to understand if they were false positives. This study involved both reading the source code, and executing attacks against each vulnerability found to understand if it was attackable (true positive) or not (false positive). This data set is further discussed in Section 3.3.3.

From this analysis, we found three main sets of attributes that led to false positives, as outlined next.

- *String manipulation*: attributes that represent PHP functions or operators that manipulate strings. These attributes are substring extraction, concatenation, addition of characters, replacement of characters, and removal of white spaces. Recall that a data flow starts at an entry point, where it is marked tainted, and ends at a sensitive sink. The taint analyzer flags a vulnerability if the data flow is not untainted by a sanitization function before reaching the sensitive sink. These string manipulation functions may result in the sanitization of a data flow, but the taint analyzer does not have enough knowledge to change the status from tainted to untainted, so if a vulnerability is flagged it may be a false positive. The combinations of functions and operators that untaint a data flow are hard to establish, so this knowledge is not simple to retrofit into the taint analyzer.

- *Validation*: a set of attributes related to the validation of user inputs, often involving an if-then-else construct. We define several attributes: data type (calls to is_int(), is_string()), is value set (isset()), control pattern (preg_match()), a test of belonging to a white-list, a test of belonging to a black-list, and error and exit functions that output an error if the user inputs do not pass a test. Similarly to what happens with string manipulations, any of these attributes can sanitize a data flow, and lead to a false positive.

- *SQL query manipulation*: attributes related to insertion of data in SQL queries (SQL injection only). We define attributes: string inserted in a SQL aggregate function (AVG, SUM, MAX, MIN, etc.), string inserted in a FROM clause, a test if the data are numeric, and data inserted in a complex SQL query. Again, any of these constructs can sanitize data of an otherwise considered tainted data flow.

For the string manipulation and validation sets, the possible values for the attributes were two, corresponding to the presence (Y) or absence (N) of at least one of these constructs in the sequence of instructions that propagates the input from an entry point to a sensitive sink. The SQL query manipulation attributes can take a third value, not assigned (NA), when the vulnerability observed is other than SQLI.

We use only two classes to classify the vulnerabilities flagged by the taint analyzer: Yes (it is a false positive), and No (it is not a false positive, but a real vulnerability). Table 3.2 shows some examples of candidate vulnerabilities flagged by the taint analyzer, one per line. For each candidate vulnerability, the table shows the values of the attributes (Y or N), and the class, which has to be assessed manually (supervized machine learning). In each line, the set of attributes forms an instance which is classified in the class. The data mining component is configured using data like this.

3.3.2 Classifiers and metrics

As already mentioned, our data mining component uses machine learning algorithms to extract knowledge from a set of labeled data. This section presents the machine learning algorithms that were studied to identify the best approach to classify candidate vulnerabilities. We also discuss the metrics used to evaluate the merit of the classifiers.

Potential vulnerability		String manipulation					Validation						SQL query manipulation				
Type	Webapp	Extract substring	String concat.	Add char	Replace string	Remove whitesp.	Type checking	IsSet entry point	Pattern control	While list	Black list	Error / exit	Aggreg. function	FROM clause	Numeric entry point	Complex query	Class
SQLI	currentcost	Y	Y	Y	N	N	N	N	N	N	N	N	Y	N	N	N	Yes
SQLI	currentcost	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	Yes
SQLI	currentcost	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	No
XSS	emoncms	N	Y	N	Y	N	N	N	N	N	N	N	NA	NA	NA	NA	Yes
XSS	Mfm 0.13	N	Y	N	Y	Y	N	N	N	N	N	N	NA	NA	NA	NA	Yes
XSS St.	ZiPEC 0.32	N	Y	N	N	N	N	N	N	N	N	N	NA	NA	NA	NA	No
RFI	DVWA 1.0.7	N	N	N	N	N	N	N	N	Y	N	Y	NA	NA	NA	NA	Yes
RFI	SAMATE	N	N	N	Y	N	N	Y	N	N	N	N	NA	NA	NA	NA	No
RFI	SAMATE	N	N	N	Y	N	N	Y	Y	N	N	N	NA	NA	NA	NA	No
OSCI	DVWA 1.0.7	N	Y	N	Y	N	N	N	N	N	Y	N	NA	NA	NA	NA	Yes
XSS St.	OWASP Vicnum	Y	N	N	N	N	N	N	Y	N	N	N	NA	NA	NA	NA	Yes
XSS	Mfm 0.13	N	N	N	N	N	N	N	N	N	Y	N	NA	NA	NA	NA	Yes

Table 3.2: Attributes and class for some vulnerabilities

Machine learning classifiers

We studied machine learning classifiers from three classes.

- *Graphical and symbolic algorithms.* This class includes algorithms that represent knowledge using a graphical model. In the ID3, C4.5/J48, Random Tree, and Random Forest classifiers, the graphical model is a decision tree. They use the information gain rate metric to decide how relevant an attribute is to classify an instance in a class (a leaf of the tree). An attribute with a small information gain has big entropy (degree of impurity of attribute or information quantity that the attribute offers to the obtention of the class), so it is less relevant for a class than another with a higher information gain. C4.5/J48 is an evolution of ID3 that does pruning of the tree, i.e., removes nodes with less relevant attributes (with a bigger entropy). The Bayesian Network is an acyclic graphical model, where the nodes are represented by random attributes from the data set.

- *Probabilistic algorithms.* This category includes Naive Bayes (NB), K-Nearest Neighbor (K-NN), and Logistic Regression (LR). They classify an instance in the class that has the highest probability. NB is a simple probabilistic classifier based on Bayes' theorem, based on the assumption of conditional independence of the probability distributions of the attributes. K-NN classifies an instance in the class of its neighbors. LR uses regression analysis to classify an instance.

- *Neural network algorithms.* This category has two algorithms: Multi-Layer Perceptron (MLP), and Support Vector Machine (SVM). These algorithms are inspired on the functioning of the neurons of the human brain. MLP is an artificial neural network classifier that maps sets of input data (values of attributes) onto a set of appropriate outputs (our class attribute, Yes or No). SVM is an evolution of MLP.

		Observed	
		Yes (FP)	No (not FP)
Predicted	Yes (FP)	True positive (tp)	False positive (fp)
	No (not FP)	False negative (fn)	True negative (tn)

Table 3.3: Confusion matrix (generic)

Classifier evaluation metrics

To evaluate the classifiers, we use ten metrics that are computed based mainly on four parameters of each classifier. These parameters are better understood in terms of the quadrants of a confusion matrix (Table 3.3). This matrix is a cross reference table where its columns are the observed instances, and its rows are the predicted results (instances classified by a classifier). Note that through all the chapter we use the terms *false positive* (FP) and *true positive* (not FP) to express that an alarm raised by the taint analyzer is incorrect (not a real vulnerability) or correct (a real vulnerability). In this section, we use the same terms, *false positive (fp)*, and *true positive (tp)*, as well as *false negative (fn)*, and *true negative (tn)*, for the output of the next stage, the FP classifier. To reduce the possibility of confusion, we use uppercase FP and lowercase *fp, tp, fn, tn* consistently as indicated.

- *True positive rate of prediction (tpp)* measures how good the classifier is: $tpp = tp/(tp + fn)$.
- *False positive rate of prediction (fpp)* measures how the classifier deviates from the correct classification of a candidate vulnerability as FP: $fpp = fp/(fp + tn)$.
- *Precision of prediction (prfp)* measures the actual FPs that are correctly predicted in terms of the percentage of total number of FPs: $prfp = tp/(tp + fp)$.
- *Probability of detection (pd)* measures how the classifier is good at detecting real vulnerabilities: $pd = tn/(tn + fp)$.
- *Probability of false detection (pfd)* measures how the classifier deviates from the correct classification of a candidate vulnerability that was a real vulnerability: $pfd = fn/(fn + tp)$.
- *Precision of detection (prd)* measures the actual vulnerabilities (not FPs) that are correctly predicted in terms of a percentage of the total number of vulnerabilities: $prd = tn/(tn + fn)$.

- *Accuracy (acc)* measures the total number of instances well classified: $acc = (tp + tn)/(tp + tn + fp + fn)$.
- *Precision (pr)* measures the actual FPs and vulnerabilities (not FPs) that are correctly predicted in terms of a percentage of the total number of cases: $pr = average(prfp, prd)$.
- *Kappa statistic (kappa)* measures the concordance between the classes predicted and observed. It can be stratified into six categories: worst, bad, reasonable, good, very good, excellent. $kappa = (po - pe)/(1 - pe)$, where $po = acc$, and $pe = (P * P' + N * N')/(P + N)^2$ to $P = (tp + fn)$, $P' = (tp + fp)$, $N = (fp + tn)$, and $N' = (fn + tn)$.
- *Wilcoxon signed-rank test (wilcoxon)* compares classifier results with pairwise comparisons of the metrics *tpp* and *fpp*, or *pd* and *pfd*, with a benchmark result of *tpp, pd* $> 70\%$, and *fpp, pfd* $< 25\%$ (Demšar, 2006).

Some of these metrics are statistical, such as rates and *kappa*, while *acc* and *pr* are probabilistic, and the last is a test.

3.3.3 Evaluation of classifiers

In this section we use the metrics to select the best classifiers for our case. Our data set has 76 vulnerabilities labeled with 16 attributes: 15 to characterize the candidate's vulnerabilities, and 1 to classify it as being false positive (Yes) or a real vulnerability (No). For each candidate vulnerability, we used a version of WAP to collect the values of the 15 attributes, and we manually classified them as false positives or not. Needless to say, understanding if a vulnerability was real or a false positive was a tedious process. The 76 potential vulnerabilities were distributed by the classes Yes, and No, with 32, and 44 instances, respectively. Figure 3.5 shows the number of occurrences of each attribute.

The 10 classifiers are available in WEKA, an open source data mining tool (Witten *et al.*, 2011). We used it for training and testing the ten candidate classifiers with a standard *10-fold cross validation* estimator. This estimator divides the data into 10 buckets, trains the classifier with 9 of them, and tests it with the 10th. This process is repeated 10 times to test every bucket, with the classifier trained with the rest. This method accounts for heterogeneities in the data set.

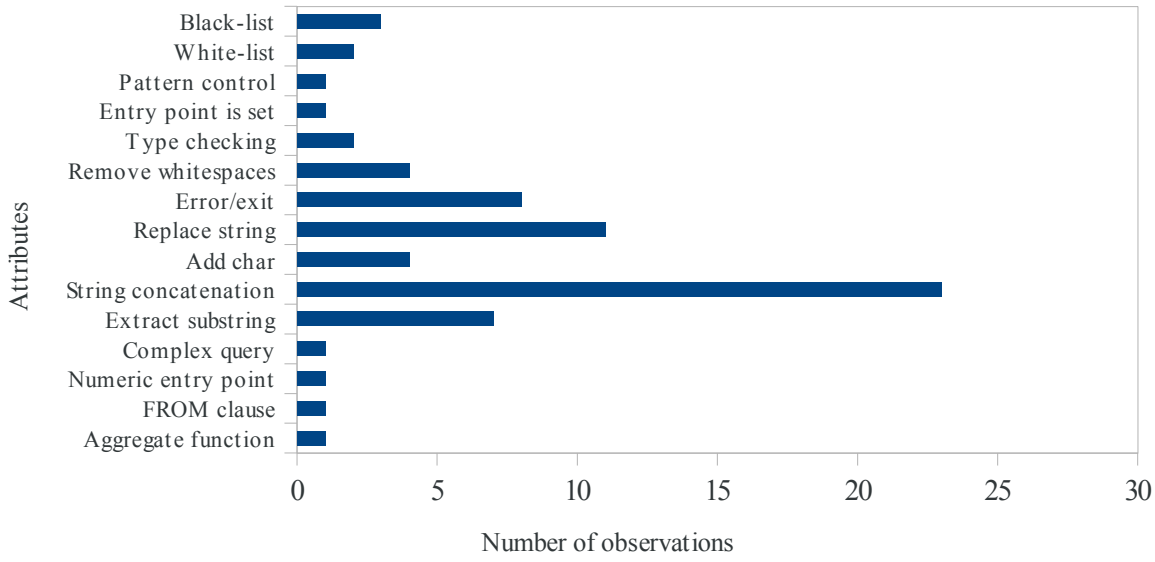


Figure 3.5: Number of attribute occurrences in the original data set.

Table 3.4 shows the evaluation of the classifiers. The first observation is the rejection of the K-NN and Naive Bayes algorithms by the Wilcoxon signed-rank test. The rejection of the K-NN algorithm is explained by the classes Yes and No not being balanced, where the first class has fewer instances, 32, than the second class, 44, which leads to unbalanced numbers of neighbors, and consequently to wrong classifications. The Naive Bayes rejection seems to be due to its naive assumption that attributes are conditionally independent, and the small number of observations of certain attributes.

Measures (%)	**ID3**	**C4.5/J48**	**Random Forest**	**Random Tree**	**K-NN**	**Naive Bayes**	**Bayes Net**	**MLP**	**SVM**	**Logistic Regression**
tpp	75.0	81.3	78.1	78.1	71.9	68.8	78.1	75.0	81.3	84.4
fpp	0.0	13.6	4.5	0.0	0.0	13.6	13.6	0.0	4.5	2.3
prfp	100.0	81.3	92.6	100.0	100.0	78.6	80.6	100.0	92.9	96.4
pd	100.0	86.4	95.5	100.0	100.0	86.4	86.4	100.0	95.5	97.7
pfd	25.0	18.8	21.9	21.9	28.1	31.3	21.9	25.0	18.8	15.6
prd	84.6	86.4	85.7	86.3	83.0	79.2	84.4	84.6	87.5	89.6
acc	89.5	82.2	88.2	90.8	82.9	78.9	82.9	89.5	89.5	92.1
(% #)	68	64	67	69	63	60	63	68	68	70
pr	91.0	84.2	88.6	92.0	86.8	78.9	82.8	91.0	89.8	92.5
kappa	77.0 very good	67.0 very good	75.0 very good	81.0 excellent	63.0 very good	56.0 good	64.0 very good	77.0 very good	78.0 very good	84.0 excellent
wilcoxon	accepted	accepted	accepted	accepted	rejected	rejected	accepted	accepted	accepted	accepted

Table 3.4: Evaluation of the machine learning models applied to the original data set

In the first four columns of the table are the decision tree models. These models select for the tree nodes the attributes that have higher information gain. The C4.5/J48 model prunes the tree to achieve better results. The branches that have nodes with weak information gain (higher entropy), i.e., the attributes with less occurrences, are removed (see Figure 3.5). However, an excessive tree pruning can result in a tree with too few nodes to do a good classification. This was what happened in our study, where J48 was the worst decision tree model. The results of ID3 validate our conclusion because this model is the J48 model without tree pruning. We can observe that ID3 has better accuracy and precision results when compared with J48: 89.5% against 82.2%, and 91% against 84.2%, respectively. The best of the tree decision models is the Random Tree. The table shows that this model has the highest accuracy (90.8% which represents 69 of 76 instances well classified) and precision (92%), and the kappa value is in accordance (81%, excellent). This result is asserted by the 100% of *prpf* that tells us that all false positive instances were well classified in class Yes; also the 100% of *pd* tells us that all instances classified in class No were well classified. The Bayes Net classifier is the third worst model in terms of kappa, which is justified by the random selection of attributes to be used as the nodes of its acyclic graphical model. Some selected attributes have high entropy, so they insert noise in the model that results in bad performance.

The last three columns of Table 3.4 correspond to three models with good results. MLP is the neural network with the best results, and curiously with the same results as ID3. Logistic Regression (LR) was the best classifier. Table 3.5 shows the confusion matrix of LR (second and third columns), with values equivalent to those in Table 3.4. This model presents the highest accuracy (92.1%, which corresponds to 70 of 76 instances well classified) and precision (92.5%), and has an excellent kappa value (84%). The prediction of false positives (first 3 rows of Table 3.4) is very good, with a great true positive rate of prediction (*tpp* = 84.6%, 27 of 32 instances), very low false alarms (*fpp* = 2.3%, 1 of 44 instances), and an excellent precision of the prediction of false positives (*prfp* = 96.4%, 27 of 28 instances). The detection of vulnerabilities (next 3 rows of the Table 3.4) is also very good, with a great true positive rate of detection (*pd* = 97.7%, 43 of 44 instances), low false alarms (*pfd* = 15.6%, 5 of 32 instances), and a very good precision of detection of vulnerabilities (*prd* = 89.6%, 43 of 48 instances).

	Observed					
	Logistic Regression		**Random Tree**		**SVM**	
Predicted	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)
Yes (FP)	27	1	25	0	56	0
No (not FP)	5	43	7	44	8	44

Table 3.5: Confusion matrix of the top 3 classifiers (first two with original data, third with a balanced data set)

Balanced data set

To try to improve the evaluation, we applied the SMOTE filter to balance the classes (Witten *et al.*, 2011). This filter doubles instances of smaller classes, creating a better balance. Figure 3.6 shows the number of occurrences in this new data set. Table 3.6 shows the results of the re-evaluation with balanced classes. All models increased their performance, and passed the Wilcoxon signed-rank test. The K-NN model has much better performance because the classes are now balanced. However, the kappa, accuracy, and precision metrics show that the Bayes models continue to be the worst. The decision tree models present good results, with the Random Tree model again the best of them, and the C4.5/J48 model still the worst. Observing Figure 3.6, there are attributes with very low occurrences that are pruned in the C4.5/J48 model. To increase the performance of this model, we remove the lowest information gain attribute (the biggest entropy attribute) and re-evaluate the model. There is an increase in its performance to 92.6% of *pr*, 93,7% of *acc*, and 85.0% (excellent) of kappa, in such a way that it is equal to the performance of the Random Tree model. Again, the neural networks and LR models have very good performance, but *SVM is the best* of the three (accuracy of 92.6%, precision of 92.3%, *prfp* and *pd* of 100%).

Measures (%)	**ID3**	**C4.5/J48**	**Random Forest**	**Random Tree**	**K-NN**	**Naive Bayes**	**Bayes Net**	**MLP**	**SVM**	**Logistic Regression**
tpp	87.3	87.5	85.9	87.5	84.4	83.6	83.6	85.9	87.5	85.9
fpp	0.0	9.1	0.0	0.0	0.0	19.5	18.2	0.0	0.0	2.3
prfp	100.0	93.3	100.0	100.0	100.0	87.5	87.5	100.0	100.0	98.2
pd	100.0	90.9	100.0	100.0	100.0	80.5	81.8	100.0	100.0	97.7
pfd	12.7	12.5	14.1	12.5	15.6	16.4	16.4	14.1	12.5	14.1
prd	84.6	83.3	83.0	84.6	81.5	75.0	76.6	83.0	84.6	82.7
acc	92.5	88.9	91.7	92.6	90.7	82.4	82.9	91.7	92.6	90.7
(% #)	99	96	99	100	98	89	89	99	100	98
pr	92.3	88.3	91.5	92.3	90.7	81.3	82.0	91.5	92.3	90.5
kappa	85.0 Excellent	77.0 Very Good	83.0 Excellent	85.0 Excellent	81.0 Excellent	64.0 Very Good	64.0 Very Good	83.0 Excellent	85.0 Excellent	81.0 Excellent
wilcoxon	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted

Table 3.6: Evaluation of the machine learning models applied to the balanced data set

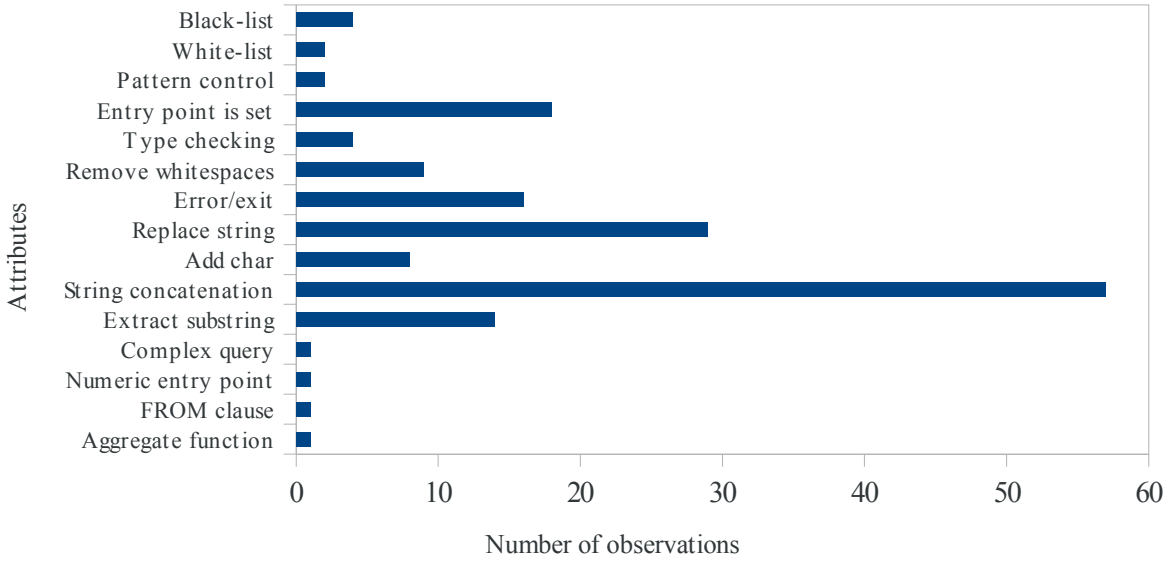


Figure 3.6: Number of attribute occurrences in the balanced data set.

Main attributes

To conclude the study of the best classifier, we need to understand which attributes contribute most to a candidate vulnerability being a false positive. For that purpose, we extracted from our data set 32 false positive instances, and classified them in three sub-classes, one for each of the sets of attributes of Section 3.3.1: string manipulation, SQL query manipulation, and validation. Then, we used WEKA to evaluate this new data set with the classifiers that performed best (LR, Random Tree, and SVM), with and without balanced classes. Table 3.7 shows the confusion matrix obtained using LR without balanced classes. The 32 instances are distributed by the three classes with 17, 3, and 12 instances. The LR performance was $acc = 87.5\%$, $pr = 80.5\%$, and $kappa = 76\%$ (very good). All 17 instances of the string manipulation class were correctly classified. All 3 instances from the SQL class were classified in the string manipulation class, which is justified by the presence of the *concatenation* attribute in all instances. The 11 instances of the validation class were well classified, except one that was classified as string manipulation. This mistake is explained by the presence of the *add char* attribute in this instance. This analysis lead us to the conclusion that the string manipulation class is the one that most contributes to a candidate vulnerability being a false positive.

		Observed		
		String manip.	SQL	Validation
Predicted	String manip.	17	3	1
	SQL	0	0	0
	Validation	0	0	11

Table 3.7: Confusion matrix of Logistic Regression classifier applied to a false positives data set

3.3.4 Selection of classifiers

After the evaluation of classifiers, we need to select the classifier that is best at classifying candidate vulnerabilities as false positives or real vulnerabilities. For that purpose, we need a classifier with great accuracy and precision, but with a rate of *fpp* as low as possible, because this rate measures the false negatives of the classifier, which is when a candidate vulnerability is misclassified as being a false positive. We want also a classifier with a low rate of *pfd*, which is when a candidate vulnerability is misclassified as being a real vulnerability. This *pfd* rate being different from zero means that source code with a false positive may be corrected, but it will not break the behavior of the application because the fixes are designed to avoid affecting the behavior of the application. Finally, we want to justify why a candidate vulnerability is classified as a false positive, i.e., which attributes lead to this classification.

Meta-models

To improve the classification performed by classifiers, our first attempt was to combine machine learning algorithms. WEKA allows us to do this using meta-models. In the evaluation made in the previous section, the Random Tree (RT) and LR were two of the best classifiers. We used the Bagging, Stacking, and Boosting algorithms with RT; and Boosting with LR (LogitBoost). The Stacking model had the worst performance with an *acc = 58%*, and thus we removed it from the evaluation. The others meta-models had in average *acc = 86.2%*, *pr = 87.7%*, *fpp = 3.8%*, and 66 instances well classified. Given these results, we concluded that the meta-models had no benefit, as they showed worst performance than RT and LR separately (see Tables 3.4, and 3.6 for these two classifiers).

Top 3 classifiers

LR was the best classifier with our original data set, but had *fpp = 2.3%* so it can misclassify candidate vulnerabilities as false positives. With the balanced data set, it was one of the best classifiers, despite *fpp* remaining unchanged. On the other hand, RT was the best decision tree classifier in both evaluations with *fpp = 0%*, i.e., no false negatives. Also, the SVM classifier was one of the best with the original data set, and the best with the balanced data set, with *fpp = 0%* unlike the *fpp = 4.5%* in the first evaluation. It was visible that SVM with the balanced data set classified correctly the two false negative instances that it classified wrongly with the original data set. Table 3.5 shows the confusion matrix for RT (4th and 5th columns), and SVM (last two columns) with no false negatives; and for LR (2nd and 3rd columns) with the number of false positives (a false positive classified as a vulnerability) lower than the other two classifiers.

Rules of induction

Data mining is typically about correlation, but the classifiers presented so far do not show this correlation. For that purpose, our machine learning approach allows us to identify combinations of attributes that are correlated with the presence of false positives, i.e., what attributes justify the classification of false positives. To show this correlation, we use induction or coverage rules for classifying the instances, and for presenting the attributes combination to that classification. For this effect, we evaluated the JRip, PART, Prism, and Ridor induction classifiers.

The results are presented in Table 3.8. Clearly, JRip was the best induction classifier, with higher *pr* and *acc*, and the only one without false negatives (*fpp = 0%*). It correctly classified 67 out of 76 instances. The instances wrongly classified are expressed by *pfd = 28.1%*. As explained, this statistic reports the number of instances that are false positives but were classified as real vulnerabilities. In our approach, these instances will be corrected with unnecessary fixes, but a fix does not interfere with the functionality of the code. So, although JRip has a higher *pfd* than the other classifiers, this is preferable to a *fpp* different from zero.

Table 3.9 shows the set of rules defined by JRip to classify our data set. The first six columns are the attributes involved in the rules, the seventh is the classification, and the last is the total number of instances covered by the rule, and the number of instances wrongly covered by the rule (the two numbers are separated by a comma). For example, the first rule (second line) classifies an instance as being false positive (Class Yes) when the *String*

Measures (%)	JRip	PART	Prism	Ridor
acc	88.2	88.2	86.8	86.8
(% #)	67	67	66	66
pr	90.0	88.5	88.4	87.5
fpp	0.0	6.8	13.6	4.5
pfd	28.1	18.6	9.7	25.0

Table 3.8: Evaluation of the induction rule classifiers applied to our original data set

concatenation and *Replace string* attributes are present. The rule covers 9 instances in these conditions, from the 32 false positives instances from our data set, none were wrongly classified (9 , 0). The last rule classifies as real vulnerability (Class No) all instances that are not covered by the previous five rules. The 44 real vulnerabilities from our data set were correctly classified by this rule. The rule classified five instances in class No that are false positives. These instances are related with *Black list* and SQL attributes, which are not cover by the other rules. This classification justifies the *pfd* value in Table 3.8. Notice that the attributes involved in this set of rules confirms the study of main attributes presented in Section 3.3.3, where the SQL attributes are not relevant, and the string manipulation and validation attributes (string manipulation first) are those that most contribute to the presence of false positives.

String concatenation	Replace string	Error / exit	Extract substring	IsSet entry point	While list	Class	Cover
Y	Y					Yes	9, 0
Y		Y				Yes	7, 0
			Y			Yes	7, 0
					Y	Yes	2, 0
	N			Y		Yes	2, 0
						No	49, 5

Table 3.9: Set of induction rules from the JRip classifier

3.3.5 Final selection and implementation

The main conclusion of our study is that there is no single classifier that is the best for classifying false positives with our data set. Therefore, we opted to use the top 3 classifiers to increase the confidence in the false positive classification. The top 3 classifiers include Logistic Regression and Random Tree trained with the original data set, and SVM trained

with the balanced data set. Also, the JRip induction rule is used to present the correlation between the attributes to justify the false positives classification. The combination of 3 classifiers is applied in sequence: first LR; if LR classifies the vulnerability as false positive, RT is applied; if false positive, SVM is applied. Only if SVM considers it a false positive is the final result determined to be a false positive. These classifiers were implemented in WAP, and trained with the original and balanced data sets as indicated.

3.4 Fixing and Testing the Source Code

3.4.1 Code correction

Our approach involves doing code correction automatically after the detection of the vulnerabilities is performed by the taint analyzer and the data mining component. The taint analyzer returns data about the vulnerability, including its class (e.g., SQLI), and the vulnerable slice of code. The code corrector uses these data to define the fix to insert, and the place to insert it. Inserting a fix involves modifying a PHP file.

A fix is a call to a function that sanitizes or validates the data that reaches the sensitive sink. Sanitization involves modifying the data to neutralize dangerous metacharacters or metadata, if they are present. Validation involves checking the data, and executing the sensitive sink or not depending on this verification. Most fixes are inserted in the line of the sensitive sink instead of, for example, the line of the entry point, to avoid interference with other code that sanitizes the variable. Table 3.10 shows the fixes, how they are inserted, and other related information.

For SQLI, the fix is inserted into the last line where the query is composed, and before it reaches the sensitive sink. However, the fix can be inserted in the line of the sensitive sink, if the query is composed there. The `san_sqli` fix applies PHP sanitization functions (e.g., `mysql_real_escape_string`), and lets the sensitive sink be executed with its arguments sanitized. The SQLI sanitization function precedes any malicious metacharacter with a backslash, and replaces others by their literal, e.g., \n by '\n'. The sanitization function applied by the `san_sqli` fix depends on the DBMS, and the sensitive sink. For example, for MySQL, the `mysql_real_escape_string` is selected if the sensitive sink `mysql_query` is reached; but for PostgreSQL, the `pg_escape_string` is used if the sensitive sink is `pg_query`. For XSS, the fixes use functions from the OWASP PHP Anti-XSS library that

Vulnerability	Fix						Output	
	Sanitization		Validation		Applied to	Function	Alarm message	Stop execution
	Addition	Substitution	Black-list	White-list				
SQLI	X	X			query	san_sqli	–	No
Reflected XSS		X			sensitive sink	san_out	–	No
Stored XSS	X	X		X	sensitive sink	san_wdata	X	No
Stored XSS		X		X	sensitive sink	san_rdata	X	No
RFI			X		sensitive sink	san_mix	X	Yes
LFI			X		sensitive sink	san_mix	X	Yes
DT /PT			X		sensitive sink	san_mix	X	Yes
SCD			X		sensitive sink	san_mix	X	Yes
OSCI			X		sensitive sink	san_osci	X	Yes
PHPCI	X	X	X		sensitive sink	san_eval	X, –	Yes, No

Table 3.10: Action and output of the fixes

replace dangerous metacharacters by their HTML entity (e.g., < becomes <). For stored XSS, the sanitization function *addslashes* is used, and the validation process verifies in run-time if an attempt of exploitation occurs, raising an alarm if that is the case. For these two classes of vulnerabilities, a fix is inserted for each malicious input that reaches a sensitive sink. For example, if three malicious inputs appear in an `echo` sensitive sink (for reflected XSS), then the `san_out` fix will be inserted three times (one per each malicious input).

The fixes for the other classes of vulnerabilities were developed by us from scratch, and perform validation of the arguments that reach the sensitive sink, using black lists, and emitting an alarm in the presence of an attack. The `san_eval` fix also performs sanitization, replacing malicious metacharacters by their HTML representation, for example backtick by &#96.

The last two columns of the table indicate if the fixes output an alarm message when an attack is detected, and what happens to the execution of the web application when that action is made. For SQLI, reflected XSS, and PHPCI, nothing is outputted, and the execution of the application proceeds. For stored XSS, an alarm message is emitted, but the application proceeds with its execution. For the others, where the fixes perform validation, when an attack is detected, an alarm is raised, and the execution of the web application stops.

3.4.2 Testing fixed code

Our fixes were designed to avoid modifying the (correct) behavior of the applications. So far, we witnessed no cases in which an application fixed by WAP started to function incorrectly,

or that the fixes themselves worked incorrectly. However, to increase the confidence in this observation, we propose using software testing techniques. Testing is probably the most widely adopted approach for ensuring software correctness. The idea is to apply a set of test cases (i.e., inputs) to a program to determine for instance if the program in general contains errors, or if modifications to the program introduced errors. This verification is done by checking if these test cases produce incorrect or unexpected behavior or outputs. We use two software testing techniques for doing these two verifications, respectively: 1) program mutation, and 2) regression testing.

Program mutation

We use a technique based on program mutation to confirm that the inserted program fixes prevent the attacks as expected. Program mutation is a form of code-based testing, as it involves using the source code of the program (Huang, 2009). This technique consists in generating variations of the program (mutants), which are afterwards used to verify if the outputs they produce differ from those produced by the unmodified program. The main idea is that, although understanding if the behavior of a program is incorrect or not is not trivial, on the contrary comparing the results of two tests of similar programs is quite feasible.

A mutant of a program P is defined as a program P' derived from P by making a single change to P (DeMillo *et al.*, 1978; T. Budd et al., 1978). Given programs P and P', and a test-case T: (A1) T *differentiates* P from P' if executions of P and P' with T produce different results; and (A2) if T fails to differentiate P from P', either P is functionally equivalent to P', or T is ineffective in revealing the changes introduced into P'. For each vulnerability it detects, WAP returns the vulnerable slice of code, and the same slice with the fix inserted, both starting in an entry point, and ending in a sensitive sink. Consider that P is the original program (that contains the vulnerable slice), and P' is the fixed program (with the fix inserted). Consider that both P and P' are executed with a test case T.

- *T differentiates P from P' (A1)*: If T is a malicious input that exploits the vulnerability in P, then P executed with T produces an incorrect behavior. P' is the fixed version of P. Therefore, if the fix works correctly, the result of the execution of P' with T differs from the result of the execution of P with T. As explained above, comparing the results of the two tests is quite feasible.

- *T does not differentiate P from P' (A2)*: If T is a benign input, and P and P' are executed with T, a correct behavior is obtained in both cases, and the result produced by both programs is equal. Input sanitization and validation do not interfere with benign inputs, so the fixes only act on malicious inputs, leaving the benign inputs untouched, and remaining the correct behavior.

Applying this approach with a large set of test cases, we can gain confidence that a fix indeed corrects a vulnerability.

Regression testing

A concern that may be raised about the use of WAP for correcting web applications is that the applications may start to function incorrectly due to the modifications made by the tool. As mentioned, we have some experience with the tool, and we never observed this problem. Nevertheless, we propose using regression testing to verify if the (correct) behavior of an application was modified by WAP. Regression testing consists in running the same tests before and after the program modifications (Huang, 2009). The objective is to check if the functionality that was working correctly before the changes still continues to work correctly.

We consider that the result of running an application test can be either *pass* or *fail*, respectively if the application worked as expected with that test case or not. We are not concerned about how the test cases are obtained. If WAP is used by the application developers, then they can simply do their own regression testing process. If WAP is employed by others, they can write their own suite of tests, or use the tests that come with the application (something that happens with many open source applications). Regression testing is successful if all the test cases that resulted in pass before the WAP modification also result in pass after inserting the fixes.

3.5 Implementation and Challenges

Implementing WAP was quite challenging for several reasons, namely the need to reduce the number of false positives/negatives, the idiosyncrasies of PHP, etc. In summary the fundamental challenges were the following:

- *Data structures*: WAP performs taint analysis navigating in the AST and propagating the taintedness through its nodes. There are two main data structures – *taint symbol*

table (TST) and *tainted execution path tree* (TEPT) – that are built while the AST is navigated, as explained in Section 3.2.

- *Interprocedural, global and context-sensitive analysis*: WAP does *global*, *interprocedural* and *context-sensitive* analysis. Interprocedural means that it analyzes the propagation of taintedness when functions are called, instead of analyzing functions in isolation. The analysis being global means that the propagation of taintedness is also propagated when functions in different modules are called. Being context-sensitive means that the result of the analysis of a function is propagated only to the point of the program where the call was made (context-insensitive analysis propagates results to all points of the program where the function is called) (Jovanovic *et al.*, 2006).

- *File name resolution*: the name of the include files has to be resolved to perform global analysis. This often involves getting the value of environment variables defined in files like `config.php` and in global, local, and array variables. WAP also handles `define` and `dirname` functions and `__DIR__` and `__FILE__` magic constants to indicate path files and/or directories to include files. These definition forms are not easy to get and track when analysing source code statically.

- *Function/method calls*: creation of TST and TEPT for each function/method call to perform interprocedural analysis. Each user function or method definition originates a clean AST. When a function/method call is performed, the corresponding AST is copied and the taint analysis is performed, navigating through the AST and creating the TST and TEPT. To perform the interprocedural analysis correctly, the taint analysis takes the current context-sensitive of the analysis already performed, i.e., the current taintedness state and propagate it through the AST. This means that WAP deals with several TST and TEPT to correctly propagate the taintedness.

- *Programming models – imperative and object oriented*: object oriented programming languages are known to be harder to analyse that imperative languages due to the use of classes, inheritance, polymorphism, etc. PHP in this sense provides the worst-of-both-worlds as it supports both programming models, which WAP has to handle. To correctly track the objects it was necessary to simulate them in memory and track their attributes and method calls in order to see the propagation of taintedness.

- *Top-down and bottom-up approaches*: WAP needs to combine both top-down and bottom-up approaches when navigating source code representations in memory. It navigates in the abstract syntax tree (AST) using the top-down approach to taint the entry points, then follows the bottom-up approach to propagate the taintedness to its parent. It identifies the vulnerable path and the right places to insert fixes using the bottom-up approach. Finally, it collects the attributes and performs the correction of the source code using the top-down approach.
- *Uncertainty of PHP syntax*: the syntax of PHP is not rigorously defined, so often the analysis of new applications breaks the parser and requires improvements.

3.6 Experimental Evaluation

WAP was implemented in Java, using the ANTLR parser generator. It has around 95,000 lines of code, with 78,500 of which generated by ANTLR. The implementation followed the architecture of Figure 3.2, and the approach of the previous sections. The evaluation presented in this section is besed on WAP version 2.1 (Medeiros, 2014).

The objective of the experimental evaluation was to answer the following questions.

1. Is WAP able to process a large set of PHP applications? (Section 3.6.1.)
2. Is it more accurate and precise than other tools that do not combine taint analysis and data mining? (Sections 3.6.2 and 3.6.3.)
3. Does it correct the vulnerabilities it detects? (Section 3.6.4.)
4. Does the tool detect the vulnerabilities that it was programmed to detect? (Section 3.6.4.)
5. Do its corrections interfere with the normal behavior of applications? (Section 3.6.5.)

3.6.1 Large scale evaluation

To show the ability of using WAP with a large set of PHP applications, we run it with 45 open source packages. Table 3.11 shows the packages that were analyzed, and summarizes the results. The table shows that more than 6,700 files and 1,380,000 lines of code were

analyzed, with 431 vulnerabilities found (at least 43 of which were false positives (FP)). The largest packages analyzed were Tikiwiki version 1.6 with 499,315 lines of code, and phpMyAdmin version 2.6.3-pl1 with 143,171 lines of code. We used a range of packages from well-known applications (e.g., Tikiwiki) to small applications in their initial versions (like PHP-Diary). The functionality was equally diverse, including for instance a small content management application like phpCMS, an event console for the iPhone (ZiPEC), and a weather application (PHP Weather). The vulnerabilities found in ZiPEC were in the last version, so we informed the programmers, who then acknowledged their existence and fixed them.

3.6.2 Taint analysis comparative evaluation

To answer the second question, we compare WAP with Pixy and PhpMinerII. To the best of our knowledge, Pixy is the most cited PHP static analysis tool in the literature, and PhpMinerII is the only tool that does data mining. Other open PHP verification tools are available, but they are mostly simple prototypes. The full comparison of WAP with the two tools can be found in the next section. This section has the simpler goal of comparing WAP's taint analyzer with Pixy, which does this same kind of analysis. We consider only SQLI and reflected XSS vulnerabilities, as Pixy only detects these two (recall that WAP detects vulnerabilities of eight classes).

Table 3.12 shows the results of the execution of the two tools with a randomly selected subset of the applications of Table 3.11: 9 open source applications, and all PHP samples of NIST's SAMATE (SAMATE, 2014). Pixy did not manage to process *mutilidae* and *WackoPicko* because they use the object-oriented features of PHP 5.0, whereas Pixy supports only those in PHP 4.0. WAP's taint analyzer (WAP-TA) detected 68 vulnerabilities (22 SQLI, and 46 XSS), with 21 false positives (FP). Pixy detected 73 vulnerabilities (20 SQLI, and 53 XSS), with 41 false positives, and 5 false negatives (FN, i.e., it did not detect 5 vulnerabilities that WAP-TA did).

Pixy reported 30 false positives that were not raised by WAP-TA. This difference is explained in part by the interprocedural, global, and context-sensitive analyses performed by WAP-TA, but not by Pixy. Another part of the justification is the bottom-up taint analysis carried out by Pixy (AST navigated from the leafs to the root of the tree), whereas the WAP-

Web application	Files	Lines of code	Analysis time (s)	Vul files	Vul found	FP	Real vul
adminer-1.11.0	45	5,434	27	3	3	0	3
Butterfly insecure	16	2,364	3	5	10	0	10
Butterfly secure	15	2,678	3	3	4	0	4
currentcost	3	270	1	2	4	2	2
dmoz2mysql	6	1,000	2	0	0	0	0
DVWA 1.0.7	310	31,407	15	12	15	8	7
emoncms	76	6,876	6	6	15	3	12
gallery2	644	124,414	27	0	0	0	0
getboo	199	42,123	17	30	64	9	55
Ghost	16	398	2	2	3	0	3
gilbitron-PIP	14	328	1	0	0	0	0
GTD-PHP	62	4,853	10	33	111	0	111
Hexjector 1.0.6	11	1,640	3	0	0	0	0
Hotelmis 0.7	447	76,754	9	2	7	5	2
Lithuanian-7.02.05-v1.6	132	3,790	24	0	0	0	0
Measureit 1.14	2	967	2	1	12	7	5
Mfm 0.13	7	5,859	6	1	8	3	5
Mutillidae 1.3	18	1,623	6	10	19	0	19
Mutillidae 2.3.5	578	102,567	63	7	10	0	10
NeoBill0.9-alpha	620	100,139	6	5	19	0	19
ocsvg-0.2	4	243	1	0	0	0	0
OWASP Vicnum	22	814	2	7	4	3	1
paCRUD 0.7	100	11,079	11	0	0	0	0
Peruggia	10	988	2	6	22	0	22
PHP X Template 0.4	10	3,009	5	0	0	0	0
PhpBB 1.4.4	62	20,743	25	0	0	0	0
Phpcms 1.2.2	6	227	2	3	5	0	5
PhpCrud	6	612	3	0	0	0	0
PhpDiary-0.1	9	618	2	0	0	0	0
PHPFusion	633	27,000	40	0	0	0	0
phpldapadmin-1.2.3	97	28,601	9	0	0	0	0
PHPLib 7.4	73	13,383	35	3	14	0	14
PHPMyAdmin 2.0.5	40	4,730	18	0	0	0	0
PHPMyAdmin 2.2.0	34	9,430	12	0	0	0	0
PHPMyAdmin 2.6.3-pl1	287	143,171	105	0	0	0	0
Phpweather 1.52	13	2,465	9	0	0	0	0
SAMATE	22	353	1	10	20	1	19
Tikiwiki 1.6	1,563	499,315	1	4	4	0	4
volkszaehler	43	5,883	1	0	0	0	0
WackoPicko	57	4,156	3	4	11	0	11
WebCalendar	129	36,525	20	0	0	0	0
Webchess 1.0	37	7,704	1	5	13	0	13
WebScripts	5	391	4	2	14	0	14
Wordpress 2.0	215	44,254	10	7	13	1	12
ZiPEC 0.32	10	765	2	1	7	1	6
Total	6,708	1,381,943	557	174	431	43	388

Table 3.11: Summary of the results of running WAP with open source packages

TA analysis is top-down (starts from the entry points, and verifies if they reach a sensitive sink).

Overall, WAP-TA was more accurate than Pixy: it had an accuracy of 69%, whereas Pixy had only 44%.

Webapp	WAP-TA				Pixy				WAP (complete)		
	SQLI	XSS	FP	FN	SQLI	XSS	FP	FN	SQLI	XSS	Fixed
currentcost	3	4	2	0	3	5	3	0	1	4	5
DVWA 1.0.7	4	2	2	0	4	0	2	2	2	2	4
emoncms	2	6	3	0	2	3	0	0	2	3	5
Measureit 1.14	1	7	7	0	1	16	16	0	1	0	1
Mfm 0.13	0	8	3	0	0	10	8	3	0	5	5
Multilidae 2.3.5	0	2	0	0	-	-	-	-	0	2	2
OWASP Vicnum	3	1	3	0	3	1	3	0	0	1	1
SAMATE	3	11	0	0	4	11	1	0	3	11	14
WackoPicko	3	5	0	0	-	-	-	-	3	5	8
ZiPEC 0.32	3	0	1	0	3	7	8	0	2	0	2
Total	22	46	21	0	20	53	41	5	14	33	47

Table 3.12: Results of running WAP's taint analyzer (WAP-TA), Pixy, and WAP complete (with data mining)

3.6.3 Full comparative evaluation

This section compares the complete WAP with Pixy and PhpMinerII. PhpMinerII does data mining of program slices that end at a sensitive sink, regardless of data being propagated through them starting at an entry point or not. PhpMinerII does this analysis to predict vulnerabilities, whereas WAP uses data mining to predict false positives in vulnerabilities detected by the taint analyzer.

We evaluated PhpMinerII with our data set using the same classifiers as PhpMinerII's authors (Shar & Tan, 2012b,c) (a subset of the classifiers of Section 3.3.2). The results of this evaluation are in Table 3.13. It is possible to observe that the best classifier is LR, which is the only one that passed the Wilcoxon signed-rank test. It had also the highest precision (*pr*) and accuracy (*acc*), and the lowest false alarm rate (*fpp* = *20%*).

The confusion matrix of the LR model for PhpMinerII (Table 3.14) shows that it correctly classified 68 instances, with 48 as vulnerabilities, and 20 as non-vulnerabilities. We can conclude that LR is a good classifier for PhpMinerII, with an accuracy of 87.2%, and a precision of 85.3%.

We now compare the three tools. The comparison with Pixy can be extracted from Table 3.12; however, we cannot show the results of PhpMinerII in the table because it does not really identify vulnerabilities. The accuracy of WAP was 92.1%, whereas the accuracy of WAP-TA was 69%, and of Pixy was only 44%. The PHPminerII results (Tables 3.13 and 3.14) are much better than Pixy's, but not as good as WAP's, which has an accuracy of

Measures (%)	C4.5/J48	Naive Bayes	MLP	Logistic Regression
tpp	94.3	88.7	94.3	90.6
fpp	32.0	60.0	32.0	20.0
prfp	86.2	75.8	86.2	90.6
pd	68.0	40.0	68.0	80.0
pfd	5.7	11.3	5.7	9.4
prd	85.0	62.5	85.0	80.0
acc	85.9	73.1	85.9	87.2
(% #)	67	57	67	68
pr	85.6	69.2	85.6	85.3
kappa	65.8 Very Good	31.7 Reasonable	65.8 Very Good	70.6 Very Good
wilcoxon	Rejected	Rejected	Rejected	Accepted

Table 3.13: Evaluation of the machine learning models applied to the data set resulting from PhpMinerII

92.1%, and a precision of 92.5% (see Table 3.4) with the same classifier.

Table 3.15 summarizes the comparison between WAP, Pixy, and PhpMinerII. We refined these values for a more detailed comparison. We obtained the intersection between the 53 slices classified as vulnerable by PHPminerII and the 68 vulnerabilities found by WAP. Removing from the 68 those found in applications that PHPminerII could not process, 37 remain, 11 of which are false positives. All the 22 real vulnerabilities detected by PHPminerII were also detected by WAP, and PHPminerII did not detect 4 vulnerabilities that WAP identified. The 11 false positives from WAP are among the 31 false positives of PHPminerII.

		Observed	
		Yes (Vul)	No (not Vul)
Predicted	Yes (Vul)	48	5
	No (not Vul)	5	20

Table 3.14: Confusion matrix of PhpMinerII with LR

Metric	WAP	Pixy	PhpMinerII
accuracy	92.1%	44.0%	87.2%
precision	92.5%	50.0%	85.2%

Table 3.15: Summary for WAP, Pixy and PhpMinerII

3.6.4 Fixing vulnerabilities

WAP uses data mining to discover false positives among the vulnerabilities detected by its taint analyzer. Table 3.12 shows that in the set of 10 packages WAP detected 47 SQLI and reflected XSS vulnerabilities. The taint analyzer raised 21 false positives that were detected by the data mining component. All the vulnerabilities detected were corrected (right-hand column of the table).

WAP detects several other classes of vulnerabilities besides SQLI and reflected XSS. Table 3.16 expands the data of Table 3.12 for all the vulnerabilities discovered by WAP. The 69 XSS vulnerabilities detected include reflected and stored XSS vulnerabilities, which explains the difference to the 46 reflected XSS of Table 3.12. Again, all vulnerabilities were corrected by the tool (last column).

Webapp	Detected taint analysis							Detected data mining	Fixed
	SQLI	RFI, LFI DT/PT	SCD	OSCI	XSS	Total	FP		
currentcost	3	0	0	0	4	7	2	5	5
DVWA 1.0.7	4	3	0	6	4	17	8	9	9
emoncms	2	0	0	0	13	15	3	12	12
Measureit 1.14	1	0	0	0	11	12	7	5	5
Mfm 0.13	0	0	0	0	8	8	3	5	5
Mutillidae 2.3.5	0	0	0	2	8	10	0	10	10
OWASP Vicnum	3	0	0	0	1	4	3	1	1
SAMATE	3	6	0	0	11	20	1	19	19
WackoPicko	3	2	0	1	5	11	0	11	11
ZiPEC 0.32	3	0	0	0	4	7	1	6	6
Total	22	11	0	9	69	111	28	83	83

Table 3.16: Results of the execution of WAP with all vulnerabilities it detects and corrects

3.6.5 Testing fixed applications

WAP returns new application files with the vulnerabilities removed by the insertion of fixes in the source code. As explained in Section 3.4.2, regression testing can be used to check if the code corrections made by WAP compromise the previously correct behavior of the application. Also, as depicted in the same section, program mutation can be used to check if the fixes correct vulnerabilities, i.e., if the new application files pass the tests made by regression testing.

For this purpose, we did regression testing using Selenium (Selenium, 2014), a framework for testing web applications. Selenium automates browsing, and verifies the results of the requests sent to web applications. The DVWA 1.0.7 application and the samples in SAMATE were tested because they contain a variety of vulnerabilities detected and corrected by the WAP tool (see Table 3.16). Specifically, WAP corrected 6 files of DVWA 1.0.7, and 10 of SAMATE.

The regression testing was carried out in the following way. First, we created in Selenium a set of test cases with benign inputs. Then, we ran these test cases with the original DVWA and SAMATE files, and observed that they passed all tests. Next, we replaced the 16 vulnerable files by the 16 files returned by WAP, and reran the tests to verify the changes introduced by the tool. The applications passed again all the tests.

3.7 Conclusions

This chapter presents an approach for finding and correcting input validation vulnerabilities in web applications, and a tool that implements the approach for PHP programs and input validation vulnerabilities. The approach and the tool search for vulnerabilities using a combination of two techniques: static source code analysis and data mining. Data mining is used to identify false positives using 3 machine learning classifiers. All classifiers were selected after a thorough comparison of several alternatives. It is important to note that this combination of detection techniques cannot provide entirely correct results. The static analysis problem is undecidable, and resorting to data mining cannot circumvent this undecidability, but only provide probabilistic results.

WAP as other tools that do taint analysis (presented in Section 2.2.1.2) also does alias analysis for detecting vulnerabilities, although it goes further by also correcting the code. Furthermore, Pixy does only module-level analysis, whereas WAP does global analysis (i.e., the analysis is not limited to a module or file, but can involve several). Contrary to our work, the works presented in Section 2.3.1 that use data mining did not aim to detect bugs and identify their location, but to assess the quality of the software in terms of the prevalence of defects and vulnerabilities. WAP is quite different because it has to identify the location of vulnerabilities in the source code, so that it can correct them with fixes. Moreover, WAP does not use data mining to identify vulnerabilities, but to predict whether the vulnerabilities found by taint analysis are really vulnerabilities or false positives.

We propose to use the output of static analysis to remove vulnerabilities automatically. A few works use approximately the same idea of first doing static analysis then doing some kind of protection, but mostly for SQL injection and XSS, and without attempting to insert fixes to correct the source code in the same way than WAP. However, saferXSS (Shar & Tan, 2012a) also corrects the source code, but differently than WAP. After to find XSS vulnerabilities, it wraps the user inputs (entry points) with functions provided by OWASP's ESAPI (OWASP, 2014a). On contrary, WAP inserts fixes in the sensitive sinks with the aim of not to compromise the behavior of the application, since it deals with several classes of vulnerabilities. However, none of these works use data mining or machine learning.

The WAP tool corrects the code by inserting fixes, i.e., sanitization and validation functions. Testing is used to verify if the fixes actually remove the vulnerabilities and do not compromise the (correct) behavior of the applications. The tool was experimented with synthetic code with vulnerabilities inserted on purpose, and with a considerable number of open source PHP applications. It was also compared with two source code analysis tools: Pixy and PhpMinerII. This evaluation suggests that the tool can detect and correct the vulnerabilities of the classes it is programmed to handle. It was able to find 388 vulnerabilities in 1.4 million lines of code. Its accuracy and precision were approximately 5% better than PhpMinerII's, and 45% better than Pixy's.

4

Detecting Vulnerabilities using *Weapons*

Static analysis tools search for vulnerabilities in source code, helping programmers to fix the code. However, these tools are programmed to detect specific sets of flaws, often SQLI and XSS (Jovanovic *et al.*, 2006; Nunes *et al.*, 2015), occasionally a few other (case of WAP and (Dahse & Holz, 2014)), and are typically hard to extend to search for new classes of vulnerabilities. Furthermore, new technologies are becoming centric in web applications. An example are NoSQL databases, particularly convenient to store big data, like the MongoDB, the most used NoSQL database (DB-Engines, 2015).

The chapter addresses the difficulty of extending these tools by proposing a modular and extensible version of the WAP tool (presented in Chapter 3), equipping it with *weapons* (WAP extensions) to detect and correct new vulnerability classes. This involves restructuring the tool in: (1) modules for the vulnerability classes that it already detects; and, more importantly, (2) a new module to be configured by the user to detect and correct new vulnerability classes without additional programming. This latter module takes as input data about the new vulnerability class: *entry points* (input sources), *sensitive sinks* (functions exploited by the attack), and *sanitization functions* (functions that neutralize malicious input). Then it automatically generates a *weapon* composed of: a *detector* to search for vulnerabilities, *symptoms* to predict false positives, and a *fix* to correct vulnerable code. We used this scheme to enhance the new version of WAP with the ability to detect 7 new classes of vulnerabilities: session fixation, header injection (or HTTP response splitting), email injection,

comment spamming injection, LDAP injection, XPath injection, and NoSQL injection.

We also demonstrate that this modularity and extensibility can be used to create weapons that deal with non-native entry points, sanitization functions, and sensitive sinks. We show this point by creating a weapon to detect SQLI vulnerabilities in WordPress (WordPress, 2015), the most popular content management system (CMS) (Imperva, 2015).

A second improvement to the tool was performed to make it more precise and accurate. We propose to increase the granularity of the analysis, adding more symptoms to the original set used in previous version of WAP and a new, larger, data set. A re-evaluation of machine learning classifiers was performed to select the new top 3 classifiers.

The version of WAP presented in this chapter is the first static analysis tool configurable to detect and correct new classes of vulnerabilities without programming. To the best of our knowledge, it is also the first static analysis tool that detects NoSQL injection and comment spamming injection (CI). The latter is currently the most exploited vulnerability in applications based on WordPress (Imperva, 2015).

The chapter is organized as follows. Section 4.1 presents briefly the architecture of the WAP tool. Section 4.2 explains the restructuring performed in the tool to make it modular and extensible. Section 4.3 presents the weapons we created to detect seven new classes of vulnerabilities and SQLI vulnerabilities in WordPress. Section 4.4 presents the experimental evaluation. The chapter ends with conclusions (Section 4.5).

4.1 Architecture

This section presents briefly the architecture of the WAP tool (detailed version in Section 3.1). WAP detects input validation vulnerabilities in PHP web applications. This version of the tool handles eight vulnerability classes: SQLI, XSS (reflected and stored), remote file inclusion (RFI), local file inclusion (LFI), directory or path traversal (DT/PT), OS command injection (OSCI), source code disclosure (SCD), and PHP command injection (PHPCI).

WAP is developed in Java, and its implementation follows the architecture of Figure 3.2, and the approach presented in Section 3.1. It is composed of 3 modules represented summarily in Figure 4.1 and explained briefly as following:

1. *Code analyzer:* parses the source code, generates an abstract syntax tree (AST), does taint analysis, and generates trees describing candidate vulnerable data-flow paths

(from an entry point to a sensitive sink). The code analyzer may return false positives as it may not recognize that certain code structures effectively sanitize data flows.

2. *False positive predictor:* obtains *symptoms* (source code features) from the candidate vulnerable data-flow paths and uses a combination of 3 classifiers to make the prediction (Logistic Regression, Random Tree, Support Vector Machine).

3. *Code corrector:* identifies the fixes to add and the places where they have to be inserted; then modifies the source code with the fixes.

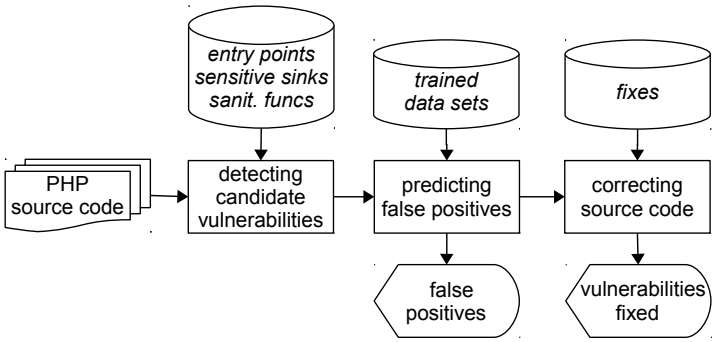


Figure 4.1: Overview of the WAP tool modules and data flow.

4.2 Restructuring WAP

We propose to extend the WAP tool to be configurable to handle new classes of input validation vulnerabilities, so we restructure the tool making it modular. We explain this process considering WAP's three original modules: code analyzer, false positive predictor, and code corrector.

4.2.1 Code analyzer

The code that does taint analysis uses three pieces of data about each class of vulnerability: entry points, sensitive sinks, and sanitization functions. Data coming from entry points is considered tainted (i.e., non-trustworthy). This component tracks how this data flows

through variables and functions, verifying if it reaches a sensitive sink. Sanitization functions block the flow of tainted data. Therefore, the taint analyzer is coded to recognize the set of functions for each vulnerability class and specific characteristics on this class (if they exist).

Restructuring the code analyzer implies, on the one hand, to reorganize the taint analyzer in sub-modules and, on the other hand, to create a generic detection sub-module configurable by the user for new vulnerability classes. The AST has to be left unmodified as it is input to all the sub-modules.

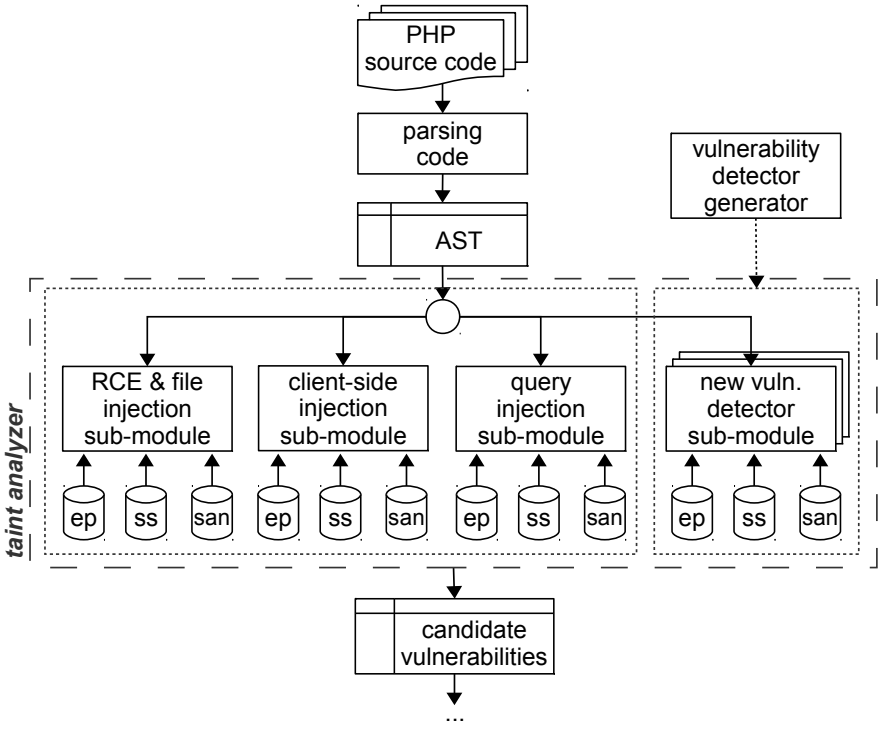


Figure 4.2: Reorganization of WAP's code analyzer module.

Figure 4.2 shows the restructured code analyzer. At the top the figure shows that PHP code is converted to an AST that is common input to all sub-modules. The sub-modules are:

1. *RCE & file injection*, dealing with vulnerabilities involving file system, files, and URLs leading to remote code execution (RCE). These vulnerabilities are OSCI, PHPCI, RFI, LFI, DT, and SCD.

2. *client-side injection*, handling vulnerabilities related with injection of client-side code (e.g., JavaScript code), namely reflected and stored XSS.

3. *query injection*, for vulnerabilities associated to queries, i.e., SQLI.

4. *vulnerability detector generator*, the generic detector configurable by the user for new vulnerabilities.

5. *new vulnerability detector sub-module*, the detectors generated by 4., one for each new vulnerability class.

Each sub-module is fed with entry points (*ep*), sensitive sinks (*ss*), and sanitization (*san*) functions. These sets of data are now stored in external files, allowing the inclusion of new items without recompiling the tool.

WAP's parser was implemented using ANTLR (Parr, 2007). This framework provides tree walkers to navigate through ASTs. The new vulnerability detector sub-module (sub-module 5.) leverages a tree walker to track data flow and understand if tainted data reaches a sensitive sink.

4.2.2 False positive predictor

The 3 classifiers of the original WAP use 15 attributes to classify the vulnerabilities found by the taint analyzer as true or false. These attributes represent 24 symptoms that may be present in source code, divided in three categories: validation, string manipulation and SQL query manipulation. Table 4.1 shows these attributes and symptoms in the two left-hand columns. The symptoms are PHP functions that manipulate entry points or variables. The attributes represent symptoms of the same kind, e.g., the *type checking* attribute represent the symptoms that check the data type of variables. Therefore, an attribute represents several symptoms. A special attribute is used to indicate the class of each instance (the 16th, last row).

We propose to improve this component in two directions: (1) by adding more symptoms to the original set used in WAP (*static symptoms*), and (2) allowing the user to define new symptoms (*dynamic symptoms*).

WAP (original)		WAPe (new version)
attribute	**symptom**	**symptom**
validation		
Type checking	is_string is_int is_float is_numeric ctype_digit	ctype_alpha is_scalar, intval, is_integer, is_long is_double, is_real ctype_alnum,
Entry point is set	isset	is_null, empty
Pattern control	preg_match	preg_match_all ereg eregi strnatcmp strcmp strncmp strncasecmp strcasecmp
White list Black list	*user functions*[1] *user functions*[2]	
Error and exit	error, exit	
Extract substring	substr	preg_split str_split explode split spliti
String concatenation	concatenation operator	implode, join
Add char	addchar	str_pad
Replace string	substr_replace str_replace preg_replace	preg_filter ereg_replace eregi_replace str_ireplace str_shuffle chunk_split
Remove whitespaces	trim	rrim, ltrim
SQL query manipulation		
Complex query Numeric entry point FROM clause Aggregated function	ComplexSQL IsNum FROM AVG, COUNT, SUM MAX, MIN	
classification		
Class	false positive (FP) real vulnerability (RV)	

[1] user functions containing white lists to validate user inputs. [2] user functions containing black lists to block user inputs

Table 4.1: Attributes and symptoms defined in the original WAP and those new. In the new WAP all symptoms are also attributes.

Static symptoms

By investigating the symptoms associated with false positives we have understood that there were several relevant symptoms not considered originally in WAP. These symptoms are listed in the right-hand column of Table 4.1. Moreover, we increased the granularity of the analysis by specifying that *all symptoms are attributes* (both old and new, 2nd and 3rd columns). Therefore, instead of 16 attributes we now have 61: 29 related to validation, 23 to string manipulation, 8 to SQL query manipulation, and the class attribute.

Modifying the attributes requires training again the classifiers and, as the number of attributes is much higher, we need also a much larger number of instances (samples of code annotated as false positive or not). The original WAP was trained with a data set of 76 instances: 32 annotated as false positives and 44 as real vulnerabilities. Each instance had 16 attributes set to 1 or 0, indicating the presence or not of symptoms for the attributes, and an attribute saying if the instance is a false positive or not. We increased the number of instances to 256, each one with 61 attributes. The instances are evenly divided in false positives and vulnerable (balanced data set). To create the data set we used WAP configured to output the candidate vulnerabilities, and we ran it with 29 open source PHP web applications. Then, each candidate vulnerability was processed manually to collect the attributes and to classify it as being a false positive or not. Finally, noise was eliminated from the data set by removing duplicated and ambiguous instances.

To perform the data mining process we used the WEKA tool (Witten *et al.*, 2011) with the original classifiers and induction rules. Also, as in the original WAP, we want a top 3 of classifiers. Our goals are that classifiers:

1. predict as many false positives correctly as possible.

2. have a fallout as low as possible (wrong classifications of vulnerabilities as false positives), avoiding to miss vulnerabilities found by the taint analyzer. This principle is important because we do not want to miss vulnerabilities found by the taint analyzer due to wrong prediction.

Table 4.2 depicts the evaluation of the three best classifiers. We adopt the same terminology defined and used in Section 3.3.2 for the first 7 metrics. The last 2 metrics are new. The last column shows the formulas to calculate each metric, based in values extracted from the confusion matrix (Table 4.3, last 2 columns).

Metrics (%)	SVM	Logistic Regression	Random Forest	Formula
tpp	94.5%	93.0%	90.6%	*tpp = recall = tp / (tp + fn)*
pfp	4.7%	4.7%	2.3%	*pfp = fallout = fp / (tn + fp)*
prfp	95.3%	95.2%	97.5%	*prfp = pr positive = tp / (tp + fp)*
pd	95.3%	95.3%	97.7%	*pd = specificity = tn / (tn + fp)*
ppd	94.6%	93.1%	91.2%	*ppd = inverse pr = tn / (tn + fn)*
acc	94.9%	94.1%	94.1%	*accuracy = (tp + tn) / N*
pr	94.9%	94.2%	94.4%	*precision = (prfp + ppd) / 2*
inform	89.8%	88.3%	88.3%	*informedness = tpp + pd -1 = tpp - pfp*
jacc	90.3%	88.8%	88.5%	*jaccard = tp / (tp + fn + fp)*

Table 4.2: Evaluation of the machine learning models applied to the data set.

Classifiers are usually selected based on accuracy and precision, but in this case the three classifiers have very similar values in both metrics: between 94% and 95%. Moreover, the compliance to goal (1) is measured by *tpp*. In terms of this metric, Support Vector Machine (SVM) had the best results and Logistic Regression (LR) the second best. In terms of goal (2), Random Forest (RF) had the best fallout rate (*pfp*). The *inform* metric expresses how the classifications made by the classifier are close to the correct (real) classifications, whereas *jacc* measures the classifications in the false positive class, taking into account false positives and negatives (Powers, 2015). For *inform*, we combine the best values of *tpp* and *pfp*, i.e., the *tpp* from SVM and the *pfp* from RF, resulting in 92%, while for *jacc* we use the correct and misclassifications of all classifiers, resulting in 92%. These measures confirm our choice of the top 3 classifiers. These classifiers are the same as those used in the original WAP, except RF that substitutes Random Tree.

The confusion matrix of these classifiers is presented in Table 4.3. SVM and LR classified incorrectly a few instances, and RF classified 3 real vulnerabilities as being false positives. Notice that this misclassification is represented as *fp* in the confusion matrix, representing the instances belonging to class *No* that were classified in class *Yes*. However, in the context of vulnerability detection this represents false negatives, i.e., vulnerabilities that were not detected.

Dynamic symptoms

We use the term dynamic symptoms to designate symptoms defined by the user that configures the tool for new vulnerabilities, whereas static symptoms are those that come

	Observed							
	SVM		Logistic Regression		Random Forest		Classifier	
Predicted	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes	No
Yes (FP)	121	6	119	6	116	3	*tp*	*fp*
No (not FP)	7	122	9	122	12	125	*fn*	*tn*

Table 4.3: Confusion matrix of the top 3 classifiers and confusion matrix notation (last two columns).

with the tool. For every dynamic symptom the user has to provide a category and a type. For example, if the user develops a function *val_int* to validate integer inputs (instead of *is_int*) he has to provide the information that the function belongs to the validation category and that it has an effect similar to the static symptom (function) *is_int*. Based on this information, the tool understands how to handle function *val_int* when predicting false positives.

Figure 4.3 presents the reorganization of the false positive predictor. When a candidate vulnerability is processed by this module: first the static and dynamic symptoms are collected from the source code; then a vector of 61 attributes is created using the map from static symptoms to attributes (stored into the tool) and the map of dynamic symptoms to attributes (created dynamically); then the vector is classified using machine learning classifiers; finally, in case of a real vulnerability, it is sent to the code corrector module to be fixed.

4.2.3 Code corrector

When a vulnerability is found, the code corrector inserts a fix that does sanitization or validation of the data flow. To make WAP modular we created two sub-modules:

1. *code fixing* sub-module, which receives the vulnerability class and the code to be fixed and inserts the fix.

2. *fix creation* that uses information and constraints provided by the user to generate a new fix for a new class of vulnerabilities.

The first does essentially what the original version of WAP already did so we focus on 2..

We propose three *fix templates* to generate automatically fixes: *PHP sanitization function*, *user sanitization*, and *user validation*. The one that is used depends on the information provided by the user.

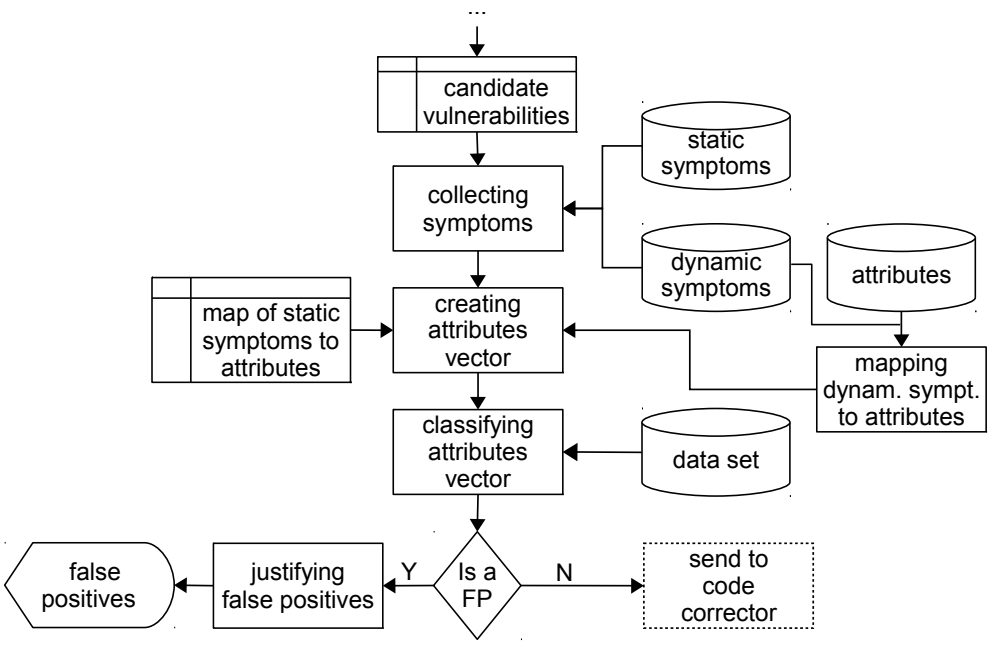


Figure 4.3: Reorganization of the false positives predictor module.

- The *PHP sanitization function* template is applied when the user specifies the PHP sanitization function used to sanitize data and the sensitive sink associated to this functions, for a given vulnerability. The sanitization function is used as fix.
- The *user sanitization* template is chosen if the user indicates the malicious characters that may be used to exploit the vulnerability and a character that can be used to neutralize them (e.g., the backslash).
- The *user validation* template is used if the user only specifies the set of malicious characters used to exploit the vulnerability. In that case the fix checks the presence of these characters, issuing a message in case there is a match.

Fixes are inserted in the line of the sensitive sink, as in the original WAP.

The Listing 4.1 shows the code of these templates. The *PHP sanitization function* template (Listing 4.1(a)) is applied when the user specifies which are the PHP sanitization functions used to sanitize data and the sensitive sink associated to this functions, for a given vulnerability. For example for SQLI vulnerability, the `mysql_real_escape_string` sanitization function was specified to the `mysql_query` sensitive sink. In the code of the figure,

for this example, the *ss_user* and *san_user* represent these two functions, respectively for SQLI vulnerability (*vulName*). The *user sanitization* template (Listing 4.1(b)) is chosen if the user indicates the malicious characters that are employed to exploit a vulnerability and which characters are used to neutralize them. For instance, for SQLI we could specify the set of malicious characters and the backslash, respectively. The code of the figure represents this set of malicious characters by the *$metachars* array and the set of characters that neutralize it by the *$replaceBy* array. To *user validation* template (Listing 4.1(c)) is utilized if the user only specifies the set of malicious characters used to exploit the vulnerability. In the code of the figure, the *$metachars* array contains the malicious characters; each one is checked to determine if it appears inside the *$input*; and in the presence of such character a message is issued.

4.2.4 Weapons

A *weapon* is a WAP extension composed by a detector, a fix and, optionally, a set of dynamic symptoms. To generate weapons we developed a *weapon generator*, external to WAP. The data needed to create a weapon is:

1. for the *detector*, the sanitization and sensitive sinks functions, plus additional entry points if they exist.

2. for the *fix*, data for the fix templates (Section 4.2.3).

3. *dynamic symptoms*, in case the user has a white/black lists of functions, or functions that do not belong to the static symptoms list (in this case, the correspondence between dynamic and static symptoms is required).

To generate a weapon, the weapon generator uses the *vulnerability detector generator* (see Section 4.2.1) that it configures with item 1. above, generating a new detector with the *ss*, *san* and *ep* files containing the data provided by the user. Next, it configures the selected fix template with item 2., generating a new fix. Then, it creates a file with item 3.. The last step is to put together the three parts, linking them to WAP. Detection is activated using a command line flag also provided by the user (e.g., *-nosqli*).

When the weapon is activated, WAP parses the code, generating an AST; next the detector navigates through the AST using the data stored in its files. The candidate vulnerabilities

```php
function san_vulName($input, $sensitiveSink){
    if (strcasecmp($sensitiveSink, "ss_user") == 0)
        return san_user($input);

    // repeat "if" statement for each pair ss_user, san_user
}
```

(a) Template of PHP sanitization function.

```php
function san_vulName($input){
    $metachars = array('m1', 'm2', ..., 'mx');
    $replaceBy = array('r1', 'r2', ..., 'rx');
    $out = str_replace($metachars, $replaceBy, $input);
    return $out;
}
```

(b) Template of user sanitization.

```php
function san_vulName($input){
    $metachars = array('m1', 'm2', ..., 'mx');
    foreach ($metachars as &metachar){
        $pos = stripos($input, $metachar);
        if ($pos != false){
            echo '<script>';
            echo 'alert("vulName attack detected!!!")';
            echo '</script>';
            return 1;
        }
    }
    return 0;
}
```

(c) Template of user validation.

Listing 4.1: Fix templates proposed.

found by the detector are processed by the false positives predictor using the symptoms defined in WAP and contained in the weapon, and the real vulnerabilities are fixed using the *code fixing* module (see Section 4.2.3) with the fix of the weapon.

4.2.5 Effort to modify WAP

Modifying WAP involved an effort with three facets:

1. making the AST independent of the navigation made by the detectors (tree walkers).

2. restructuring the code to create the three sub-modules for the vulnerabilities originally considered (Section 4.2.1), to integrate the dynamic symptoms (Section 4.2.2), and to make the code corrector able to receive new fixes (Section 4.2.3).

3. coding the *weapon generator* module (Section 4.2.4).

From the three facets, 3. was the one that required more effort. We had to build a new java package to create weapons (*new vulnerability detector sub-module*), a frontend for the user to configure the weapon generator, templates to create automatically fixes, and to integrate the weapon in WAP. When the weapon generator is executed it creates a new java package and compiles it, building a *jar* to be integrated with the WAP tool.

4.3 Extending WAP with weapons

This section presents how we extended WAP for seven new vulnerabilities classes, as well as how this extension was done. The seven vulnerability classes are the following (6 were presented in Section 2.1): LDAP injection (LDAPI), XPath injection (XPathI), NoSQL injection (NoSQLI), comment spamming (CS), header injection or HTTP response splitting (HI), email injection (EI), and the seventh vulnerability class is the session fixation (SF). With the exception of SF, all of them are input validation vulnerabilities, meaning that they are created by lack of sanitization or validation of user inputs (entry points) before they reach a sensitive sink.

Session fixation allows an attacker to force a web client to use a specific ("fixed") session ID, allowing him to access the account of the user. Avoiding this vulnerability is not trivial as there is no sanitization function to apply or set of malicious characters to recognize. A way to defend against SF is to avoid using a session token provided by the user (OWASP, 2013; Scambray *et al.*, 2011).

The section also presents the extension to detect SQLI in WordPress plugins that uses WordPress functions as entry points, sanitization functions, and sensitive sinks.

To demonstrate how we can take advantage of the modularity we created in WAP, we opted by extending it in two different ways: reusing the sub-modules presented in Section 4.2.1 and with weapons (Section 4.3.2). However, a normal user would probably use the second form.

4.3.1 Reusing the sub-modules

The detection of four of the vulnerabilities referenced above can be integrated in the sub-modules of Section 4.2.1 and the fixes to remove them can be created using a fix template (Section 4.2.3). Table 4.4 shows the classes of vulnerabilities integrated in each sub-module and the sensitive sinks added to detect each vulnerability. These functions were inserted in the *ss* file of each sub-module. No sanitization functions or entry points were added to the *san* and *ep* files.

Sub-module	Vuln.	Sensitive sink
RCE & file injection	SF	setcookie, setdrawcookie, session_id
client-side injection	CS	file_put_contents, file_get_contents
query injection	LDAPI XPathI	ldap_add, ldap_delete, ldap_list, ldap_read, ldap_search xpath_eval, xptr_eval, xpath_eval_expression

Table 4.4: Sensitive sinks added to the WAP sub-modules to detect new vulnerability classes.

In relation to LDAPI and XPathI, a fix was created for each one using the *user validation* fix template. For CS we changed WAP's `san_read` and `san_write` fixes. These fixes deal with the sensitive sinks specified above for the CS vulnerability. They validate the user inputs contents against JavaScript code, so we changed them to also check the input contents against URIs/hyperlinks. For SF we created a fix from scratch.

4.3.2 Creating weapons

We used the scheme presented in Section 4.2.4 to create three weapons, for (1) NoSQLI, (2) HI and EI, and (3) SQLI for WordPress.

NoSQLI weapon

NoSQL is a common designation for non-relational databases used in many large-scale web applications. There are various NoSQL database models and many engines that implement them. MongoDB (MongoDB, 2015) is the most popular engine implementing the document store model (DB-Engines, 2015). Therefore, we opted for creating a weapon to detect NoSQLI in PHP web applications that connect to MongoDB. We configured the weapon generator with: (1) the `find`, `findOne`, `findAndModify`, `insert`, `remove`, `save` and

`execute` sensitive sinks and the `mysql_real_escape_string` sanitization function; (2) the *PHP sanitization* fix template to sanitize the user inputs that reach that sink with that sanitization function, resulting in the *san_nosqli* fix; and (3) no dynamic symptoms. The weapon is activated by the *-nosqli* flag.

HI and EI weapon

We configured the weapon generator with: (1) the `header` and `mail` sensitive sinks and no sanitization functions; (2) the *user sanitization* fix template to check the malicious characters presented in Section 2.1 and to replace them by a space, resulting in the *san_hei* fix; and (3) no dynamic symptoms. The weapon is activated with the *-hei* flag of WAP.

SQLI for WordPress weapon

WordPress has a set of functions that sanitize and validate different data types, which are used in some add-ons. It has also its own sinks to handle SQL commands (*$wpdb* class). If we want to analyze, for example, WordPress plugins with WAP for SQLI vulnerabilities, we need a weapon that recognizes these functions. Therefore, we configured the weapon generator with: (1) the sensitive sinks and sanitization functions from *$wpdb*; (2) the *PHP sanitization* fix template to sanitize the user inputs that reach those sinks with those sanitization functions, resulting in the *san_wpsqli* fix; and (3) dynamic symptoms, with validation functions from *$wpdb* and their corresponding static symptoms. The weapon is activated by the flag *-wpsqli*.

4.4 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions:

1. Is the new version of WAP able to detect the new vulnerabilities (Section 4.4.1 and Section 4.4.2)?
2. Does it remain able to detect the same vulnerabilities as the previous version of WAP (Section 4.4.1)?
3. Is the new version it more accurate and precise in predicting false positives (Section 4.4.1)?

4. Can it be equipped with weapons configured with non-native PHP functions and detect vulnerabilities (Section 4.4.2)?

For convenience, in this section we designate the new version of the WAP tool by *WAPe*.

4.4.1 Real web applications

To assess the new version of the tool and to answer the first three questions, we run WAPe with 54 web application packages written in PHP and compare it with the prior version of the tool.

WAPe analyzed a total of 8,374 files corresponding to 2,065,914 lines of code of the 54 packages. It detected 413 real vulnerabilities from several classes in 17 applications, in which 366 of them are zero-day vulnerabilities. The largest packages analyzed were *Play sms v1.3.1* and *phpBB v3.1.6_Es* with 248,875 and 185,201 lines of code. Table 4.5 summarizes this analysis presenting the 17 packages where these vulnerabilities were found and some information about the analysis. These 17 packages contain 4,714 files corresponding to 1,196,702 lines of code. The total execution time for the analysis was 123 seconds, with an average of 7.2 seconds per application. This average time indicates that the tool has a good performance as it searches for 15 vulnerability classes in one execution.

We run the same 54 packages with the old version of WAP. The tool flagged as vulnerable the same 15 applications (less 2 packages, since they only contain classes of vulnerabilities not known by WAP). Table 4.6 presents the detection made by the two tools distributed by the 10 classes of vulnerabilities and the false positives predicted and not predicted. The third to sixth columns show the number of real vulnerabilities that the tools found for the classes that both detect, i.e., the 386 vulnerabilities of classes SQLI, XSS, RFI, LFI, DT and SCD; 340 of them are zero-day vulnerabilities. This provides a positive answer to the second question: WAPe still discovers the vulnerabilities detected by the old WAP.

The next four columns correspond to the new vulnerabilities that WAPe was equipped to detect and the following column is the total of vulnerabilities detected by WAPe (413 vulnerabilities). WAPe detected 26 zero-day vulnerabilities of the LDAPI, HI, and CS classes, plus one known SF vulnerability. The vulnerabilities found in the *Pivotx v2.3.10* and *refbase v0.9.6* (for XSS) packages were previously discovered and registered in Packet storm (Packet storm, 2015) and CVE-2015-7383. The *Community Mobile Channels v0.2.0* application was the most vulnerable mobile application with 47 vulnerabilities (SQLI and XSS mostly). This

Web application	Version	Files	Lines of code	Analysis time (s)	Vuln. files	Vuln. found
Admin Control Panel Lite 2	0.10.2	14	1,984	1	9	81
Anywhere Board Games	0.150215	3	501	1	1	3
Clip Bucket	2.7.0.4	597	148,129	11	16	22
Clip Bucket	2.8	606	149,830	12	18	26
Community Mobile Channels	0.2.0	372	119,890	8	116	47
divine	0.1.3a	5	706	1	2	9
Ldap address book	0.22	18	4,615	2	4	1
Minutes	0.42	19	2,670	1	2	10
Mle Moodle	0.8.8.5	235	59,723	18	4	7
Php Open Chat	3.0.2	249	83,899	7	9	11
Pivotx	2.3.10	254	108,893	6	1	1
Play sms	1.3.1	1,420	248,875	19	7	6
RCR AEsir	0.11a	8	396	1	6	13
refbase	0.9.6	171	109,600	10	18	48
SAE	1.1	150	47,207	7	39	48
Tomahawk Mail	2.0	155	16,742	3	3	3
vfront	0.99.3	438	93,042	15	25	77
	Total	4,714	1,196,702	123	280	413

Table 4.5: Summary of results for the new version of WAP with real web applications.

seems to confirm the general impression that the security of mobile applications is not always the best. Also interesting is the fact that the most recent version of *Clip Bucket* contains more 4 SQLI and the same 22 vulnerabilities than the previous version.

WAP reported more vulnerabilities than WAPe, but they were false positives. The last four columns of the table show the number of false positives predicted (FPP) and not predicted (FP) by WAP (the first two columns) and WAPe (the next two columns). The original tool correctly predicted 62 false positives and incorrectly 60 as not being so. WAPe predicted 104 false positives: the same as WAP plus 42 that WAP classified as not being false positives. This means that the data mining improvements proposed in this chapter made the tool more accurate and precise in prediction of false positives and detection of real vulnerabilities.

We analyzed the 18 cases reported by WAPe as not being false positives; some of them had function calls that we did not consider as symptoms, such as calls to functions *sizeof* and *md5*, whereas others contained sanitization functions developed by the applications' programmers. For example, the *vfont v0.99.3* application contains 6 of these cases, using a function named *escape* to sanitize the user inputs. To demonstrate the extensibility of the tool for such functions, we fed it with that non-native PHP function (*escape*) as being an external sanitization function and belonging to the sanitization list (see Section 4.2.1), and we run the

Web application	Version	WAP & WAPe real vuls.				WAPe real vuls.					WAP FP		WAPe FP	
		SQLI	XSS	Files*	SCD	LDAPi	SF	HI	CS	Total	FPP	FP	FPP	FP
Admin Control Panel Lite 2	0.10.2	9	72							81	8		8	
Anywhere Board Games	0.150215		1	1					1	3				
Clip Bucket	2.7.0.4		10	11				1		22	2	4	6	
Clip Bucket	2.8	4	10	11				1		26	2	4	6	
Community Mobile Channels	0.2.0	14	27		3			3		47	4		4	
divine	0.1.3a		4	2					3	9				
Ldap address book	0.22					1				1				
Minutes	0.42	9						1		10				
Mle Moodle	0.8.8.5		6	1						7	2	1	2	1
Php Open Chat	3.0.2		10			1				11				
Pivotx	2.3.10						1			1	9		9	
Play sms	1.3.1		6							6		2	2	
RCR AEsir	0.11a		9	3					1	13	1		1	
Refbase	0.9.6		46					2		48	7	4	11	
SAE	1.1	11	25	10	1			1		48		23	12	11
Tomahawk Mail	2.0	2	1							3	1	2	3	
vfront	0.99.3	23	28	16				10		77	26	20	40	6
	Total	72	255	55	4	2	1	19	5	413	62	60	104	18

*DT & RFI, LFI vulnerabilities

Table 4.6: Vulnerabilities found and false positives predicted and reported by the two versions of WAP in web applications.

tool again for that application. The tool correctly did not report these 6 cases. We recall that WAP does not report candidate vulnerabilities that are sanitized. This example shows that a user can configure WAPe for a specific web application during its development, feeding WAPs with user functions developed for that application and helping the user revising the code of the application.

4.4.2 WordPress plugins

To answer the first and last questions, and to find previously-unknown (zero-day) vulnerabilities, we run WAPe with a set of 115 WordPress (WP) plugins (WordPress, 2015), 5 of which with vulnerabilities registered in CVE (CVE, 2015).

WordPress is the most adopted CMS and supports plugins developed by many different teams. We selected 115 plugins from different tags (arts, food, health, shopping, travel, authentication, popular plugins and others) and distributed by several ranges of downloads, from less than 2000 to more than 500K. The popular plugins fit in this last range, having

Plugin	Version	Real vulnerabilities						Total	FPP	FP
		SQLI	XSS	Files*	SCD	CS	HI			
Appointment Booking Calendar**	1.1.7	1	3					4	1	
Auth0	1.3.6		1					1		
Authorizer	2.3.6		2					2		
BuddyPress	2.4.0							0	1	
Contact formgenerator	2.0.1	11						11		
CP Appointment Calendar	1.1.7	2						2		
Easy2map**	1.2.9		1	2				3		
Ecwid Shopping Cart	3.4.6		1					1		
Gantry Framework	4.1.6		3					3		
Google Maps Travel Route	1.3.1	1	2					3		
Lightbox Plus Colorbox	2.7.2		8					8		
Payment form for Paypal pro**	1.0.1		2					2		
Recipes writer	1.0.4		4					4		
ResAds**	1.0.1		2					2		
Simple support ticket system**	1.2	18						18		
The CartPress eCommerce Shopping Cart	1.4.7	8	17					25		
WebKite	2.0.1	1						1		
WP EasyCart - eCommerce Shopping Cart	3.2.3	13	6	29	5	2	5	60		
WP Marketplace	2.4.1		9					9		1
WP Shop	3.5.3		5					5	1	
WP ToolBar Removal Node	1839		1					1		
WP ultimate recipe	2.5							0		1
WP Web Scraper	3.5		3					3		
	Total	55	71	31	5	2	5	169	3	2

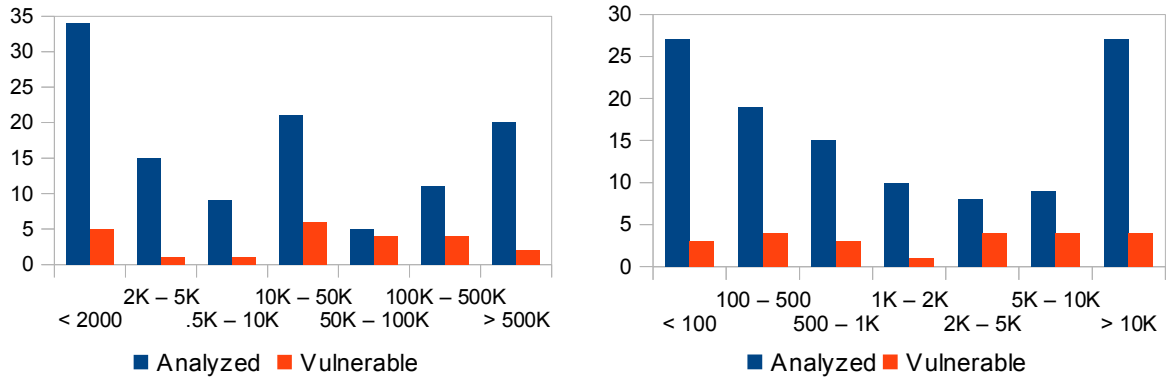
*DT & RFI, LFI vulnerabilities

**plugins with vulnerabilities registered in CVE-2015-7319, CVE-2015-7320, CVE-2015-7666, CVE-2015-7667, CVE-2015-7668, CVE-2015-7669, CVE-2015-7670

Table 4.7: Vulnerabilities found by new version of WAP in WordPress plugins.

some of them more than 1M downloads. Figure 4.4(a) shows the number of downloads of these plugins and Figure 4.4(b) the number of web sites that have these plugins active.

WAPe discovered 153 zero-day vulnerabilities and detected 16 known vulnerabilities. Table 4.7 shows the 23 plugins with vulnerabilities, distributed by 8 classes. The *wpsqli* weapon detected 55 SQLI vulnerabilities, while the other detectors found the remaining 114 vulnerabilities of the XSS, RFI, LFI, DT, HI and CS classes (last 2 are new). For the known 5 vulnerable plugins (*appointment-booking-calendar 1.1.7*, *easy2map 1.2.9*, *payment-form-for-paypal-pro 1.0.1*, *resads 1.0.1* and *simple-support-ticket-system 1.2*), we confirmed the vulnerabilities using the information about them published in BugTraq (BugTraq, 2015). However, for the *simple-support-ticket-system 1.2* plugin WAPe detected more 13 SQLI vulnerabilities than those that were registered.



(a) Downloads

(b) Active installs

Figure 4.4: Downloads and active installed plugins of 115 analyzed (blue columns) and 23 vulnerable (orange columns) plugins.

The 23 plugins fit in all ranges of downloads, as depicted by the orange columns of Figure 4.4(a). 16 of them have more than 10K downloads, reaching more than 500K downloads. All ranges of active WP installations contain vulnerable plugins, as shown by the orange columns of Figure 4.4(b). 12 plugins are used in more than 2000 web sites. The vulnerable *Lightbox Plus Colorbox* plugin is active in more than 200,000 web sites (the most used plugin), making these web sites vulnerable to XSS attacks.

Figure 4.5 presents the vulnerabilities detected by class for the 17 web applications and 23 WP plugins. Clearly SQLI and XSS continue to be the most prevalent classes. Moreover, it is possible to observe that WAPe detects correctly the vulnerabilities it was extended to detect. In both analysis it detected HI and CS vulnerabilities, while LDAPI and SF were only detected in the web applications (not plugins).

All these vulnerabilities were reported to the developers of the web applications and WP plugins. Some already confirmed their existence. All were confirmed by us manually.

4.5 Conclusions

The chapter presents the extension of the WAP tool to detect new vulnerabilities. It addresses the difficulty of extending these tools by proposing a modular and extensible version of the WAP tool, equipping it with “weapons” to detect (and correct) vulnerabilities of new classes. The approach involved restructuring WAP to make it modular and the creation of a new

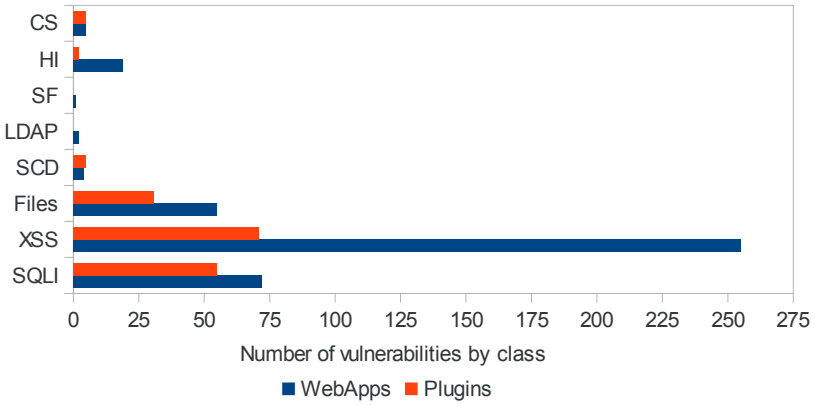


Figure 4.5: Number of vulnerabilities detected by class in the vulnerable web applications and WordPress plugins.

module to generate weapons, i.e., to generate automatically detectors and fixes to detect and remove new classes of vulnerabilities. To predict false positives the precision and accuracy of the data mining process has been improved, adding more symptoms about false positives and instances.

The new version of the tool was evaluated with seven new vulnerability classes using 54 web application packages and 115 WordPress plugins, adding up to more than 8,000 files and 2 million lines of code. The tool discovered respectively 366 and 153 zero-day vulnerabilities, i.e., 519 previously-unknown vulnerabilities. In our experiments our modular and extensible tool has shown a much higher ability to detect new (zero-day) vulnerabilities than the original version.

5

Learning to Detect Vulnerabilities

Programmers often use *static analysis tools* to search for vulnerabilities automatically in the application source code, then removing them. However, developing these tools requires explicitly coding knowledge about how each vulnerability is detected (Dahse & Holz, 2014; Fonseca & Vieira, 2014; Jovanovic *et al.*, 2006), which is complex. Moreover, this knowledge may be wrong or incomplete, making the tools inaccurate (Dahse & Holz, 2015). For example, if the tools do not understand that a certain function sanitizes inputs, this could lead to a false positive (a warning about an inexistent vulnerability).

This chapter presents a new approach for static analysis, leveraging classification models for sequences of observations that are commonly used in the field of natural language processing (NLP). Currently, NLP tasks such as parts-of-speech tagging or named entity recognition are typically modeled as sequence classification problems, in which a class (e.g., a given morpho-syntactic category) is assigned to each word in a given sentence, according to estimates given by a structured prediction model that takes word order into consideration. The model's parameters (e.g., symbol emission and class transition probabilities, in the case of hidden Markov models) are typically inferred using supervised machine learning techniques, leveraging annotated corpora.

We propose applying the same approach to programming languages. These languages are artifical but they have many characteristics in common with natural languages, such as the existence of words, sentences, a grammar, and syntactic rules. NLP usually employs machine learning to extract rules (knowledge) automatically from a *corpus*. Then, with this knowledge, other sequences of observations can be processed and classified. NLP has to

take into account the *order* of the observations, as the meaning of sentences depends on this order. Therefore it involves forms of classification more sophisticated than classification based on *standard classifiers* (e.g., naive Bayes, decision trees, support vector machines) that simply verify the presence of certain observations, without considering any order and relation between them.

This work is the first to propose an approach in which *static analysis tools learn to detect vulnerabilities automatically using machine learning*. The approach involves using machine language techniques that take the order of source code instructions into account – *sequence models* – to allow accurate detection and identification of the vulnerabilities in the code.

We specifically use a *hidden Markov model* (HMM) (Rabiner, 1989) to characterize vulnerabilities based on a set of source code slices with their *code elements* (e.g., function calls) annotated as tainted or not, taking into consideration the code that validates, sanitizes, and modifies inputs. The model can then be used as a static analysis tool to discover vulnerabilities in source code. A HMM is a Bayesian network composed of nodes representing states and edges representing transitions between states. In a HMM the states are hidden, i.e., are not observed. Given a sequence of observations, the hidden states (one per observation) are discovered following the HMM, taking into account the order of the observations. The HMM can be used to find the sequence of states that *best* explains the sequence of observations (of code elements, in our case). To detect vulnerabilities we introduce the idea of revealing the discovered hidden states of the code elements that compose the slice. This is interesting because the state of the elements determines if they are *tainted*, i.e., if the state may have been defined by an input, which may have been provided by an adversary. This allows the tool to interpret the execution of the slice statically, i.e., without actually running it. Notice that transitioning from a state to another requires understanding how the code elements behave in terms of sanitization, validation and modification, or if they affect the data flow somehow. This understanding is performed by the machine learning algorithm we propose.

The chapter also presents the DEKANT tool – *hidDEn marKov model diAgNosing vulnerabiliTies* – that implements our approach. DEKANT first extracts slices from the source code, next translates these slices into an intermediate language – *intermediate slice language* (ISL) – and retrieves their variable map. Then it analyses that representation, with the assistance of its variable map, to understand if there are vulnerabilities or not. Finally, the tool outputs the vulnerabilities, identifying them in the source code.

The chapter is organized as follows: the next section (Section 5.1) gives an overview of the approach for detection input validation vulnerabilities in web applications using static analysis based in a sequence model that learns to classify. Then, the Intermediate Slice Language (ISL) is characterized and described in Section 5.2. The ISL is used to translate PHP slices in a tokenized language, more simple to process by a sequence model. Section 5.3 presents the model that receives the translated PHP slices to classify them as being vulnerable or not, detecting thus vulnerabilities. In Section 5.4 the DEKANT tool that implements the model is presented and in Section 5.5 an experimental evaluation is showed. The chapter ends with discussion, including with related work, and conclusions (Sections 5.6 and 5.7).

5.1 Overview of the Approach

The approach has two phases: *learning* and *detection*. In the first, an annotated data set is used to acquire knowledge about vulnerabilities. In the second, vulnerabilities are detected using a sequence model, a HMM.

The HMM captures how calls to sanitization functions, validation and string modification affect the data flows between entry points and sensitive sinks. These factors may lead state to change from not tainted to tainted or vice-versa. However, we do not tell the model how to understand these functions, but train it automatically using the annotated data set (see Section 5.3).

The two phases are represented in Figure 5.1. The *learning phase* is executed when the corpus is first defined or later modified and is composed of the following sequence of steps:

1. *Building the corpus:* to build the corpus with a set of source code slices annotated either as vulnerable or non-vulnerable, to characterize code with flaws and code that handles inputs adequately (see Section 5.3.1). Duplicates have to be removed.

2. *Knowledge extraction:* to extract knowledge from the corpus (the *parameters* of the model) and represent it with probability matrices (see Section 5.3.2).

3. *Training HMM:* to train the HMM to characterize vulnerabilities with knowledge contained in the *parameters*.

The *detection phase* is composed of the following steps:

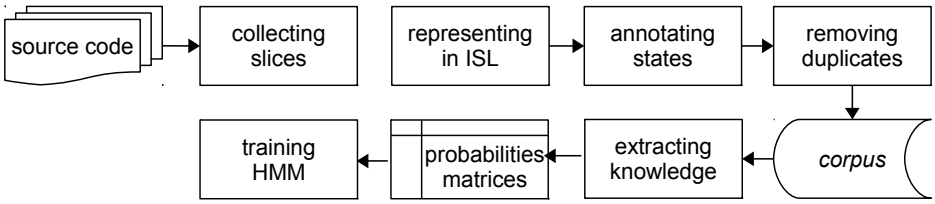
1. *Slice extraction:* to extract slices from the source code, with each slice starting in an entry point and finishing in a sensitive sink. This is done by the *slice extractor*, which tracks the entry points and their dependencies until they reach a sensitive sink, independently if they are sanitized, validated and/or modified. The resulting slice is a sequence of tracked instructions.

2. *Slice translation:* to translate the slice into *Intermediate Slice Language* (ISL). We designate the slice in ISL by *slice-isl*. During this translation, a *variable map* is created containing the variables present in the slice source code. ISL is a categorized language with grammar rules that aggregate in categories the functions of the server-side language by their functionality.

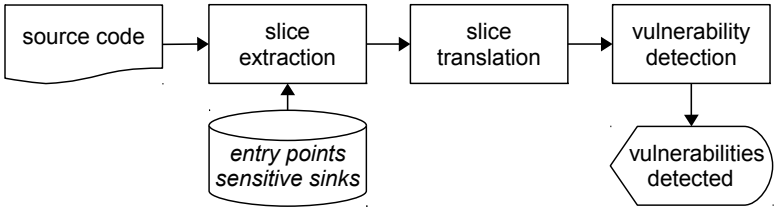
3. *Vulnerability detection:* to use the HMM to find the best sequence of states that explains *slice-isl*. Each *slice-isl* instruction (sequence of observations) is classified by the model after the tainted variables from the previous instruction determine which emission probabilities will be selected for the instruction to be classified. The classification of the last observation from the last instruction of the *slice-isl* will classify the whole slice as containing a vulnerability or not. If a vulnerability is detected, its description (including its location in the source code) is reported.

5.2 Intermediate Slice Language

As explained, slices are translated into ISL. All slices begin with an entry point and end with a sensitive sink; between them there can be other entry point assignments, input validations, sanitizations, modifications, etc. A slice contains all instructions (lines of code) that manipulate an entry point and the variables that depend on it, but no other instructions. These instructions are composed of *code elements* (e.g., entry points, variables, functions) that are categorized in *classes* of elements with the same purpose (e.g., class `input` contains PHP entry points like `$_GET` and `$_POST`). The classes are the *tokens* of the ISL language. ISL is essentially a representation of the instructions in terms of these classes. Therefore, the representation of a slice in ISL is an abstraction of the original slice, which is simpler to



(a) Learning phase.



(b) Detection phase.

Figure 5.1: Overview on the proposed approach.

process. Next we present the ISL, assuming the language of the code inspected is PHP, but the approach is generic and other languages could be considered.

5.2.1 ISL tokens and grammar

Tokens

To define the ISL tokens, we studied which PHP code elements could manipulate entry points and be associated to vulnerabilities or prevent them (e.g., functions that do sanitization or replace characters in strings). Moreover, we examined many slices (vulnerable and not) to check the presence of these code elements. The code elements representing PHP functions were carefully studied to understand which of their parameters are relevant for vulnerability detection. Some code elements are represented by more than one token. For instance, the *mysql_query* function and its parameter are represented by two tokens: `ss` (sensitive sink) and `var` (variable; or `input` if the parameter is an entry point).

Table 5.1 shows the 22 ISL tokens (column 1). The first 20 represent code elements and their parameters, whereas the last two are specific for the corpus and the implementation of the model (see Sections 5.3 and 5.4). Each of the 20 tokens represents one or more PHP functions. Column 2 says the purpose of the functions and column 3 gives function examples.

Column 4 defines the taintdness status of each token, which is used to build the corpus (see Section 5.3.1).

Some remarks on some tokens:

- `cond` corresponds to an `if` statement with validation functions over variables (user inputs) from the slice. This token allows the correlation and verification of the relation between the validated variables and the variables that appear inside the `if` branches.
- `char5` and `char6` represent the amount of characters from a string manipulated by functions that extract or replace the user input contents.
- `start_where` represents the place in the string (begin, middle or end) where the user input contents suffers modifications by functions that extract or replace characters.
- `var_vv` represents variables with `Taint` state, i.e., that are tainted, meaning that they have values that depend on entry points. This token is used in the corpus and emitted by the model, but is not used when a slice is translated to ISL. The reason is that ISL is used to represent PHP slices, before anything is known about tainted variables or vulnerabilities. Therefore, ISL represents variables by `var` indicating a variable without state or with `N-Taint` state (a variable not tainted or untainted). On the other hand, the corpus contains information about vulnerabilities, therefore it must contain tainted variables which are represented by this token. The model emits this token when a variable represented by `var` is classified with `Taint` state (see Section 5.3).
- `miss`, also used in the corpus, serves to normalize the length of sequences (see Section 5.4).

There are some code elements whose token representation depends on the context in which they appear in the source code – *context-sensitive*. The `char5` and `char6` tokens are two of such cases. They are correctly represented in ISL if the amount of manipulated characters is explicitly specified in the source code by an integer value. However, if the amount is calculated using a mathematical expression, obtaining their value in runtime, we are dealing with an unspecified case. These cases may originate false positives or false negatives, but we prefer to define a model generating some false positives than false negatives, i.e., to report some non-existent vulnerabilities than to miss some vulnerabilities that exist in the source code. Therefore in case the number of characters is undefined, ISL represents it by the

Token	Description	PHP Function	Taint
input	entry point	$_GET, $_POST, $_COOKIE, $_REQUEST $_HTTP_GET_VARS, $_HTTP_POST_VARS $_HTTP_COOKIE_VARS, $_HTTP_REQUEST_VARS $_FILES, $_SERVERS	Yes
var	variable	–	No
sanit_f	sanitization function	mysql_escape_string, mysql_real_escape_string mysqli_escape_string, mysqli_real_escape_string mysqli_stmt_bind_param, mysqli::escape_string mysqli::real_escape_string, mysqli_stmt::bind_param htmlentities, htmlspecialchars, strip_tags, urlencode	No
ss	sensitive sink	mysql_query, mysql_unbuffered_query, mysql_db_query mysqli_query, mysqli_real_query, mysqli_master_query mysqli_multi_query, mysqli_stmt_execute, mysqli_execute mysqli::query, mysqli::multi_query, mysqli::real_query mysqli_stmt::execute fopen, file_get_contents, file, copy, unlink, move_uploaded_file imagecreatefromgd2, imagecreatefromgd2part, imagecreatefromgd imagecreatefromgif, imagecreatefromjpeg, imagecreatefrompng imagecreatefromstring, imagecreatefromwbmp imagecreatefromxbm, imagecreatefromxpm require, require_once, include, include_once readfile passthru, system, shell_exec, exec, pcntl_exec, popen echo, print, printf, die, error, exit file_put_contents, file_get_contents eval	Yes
typechk_str	type checking string function	is_string, ctype_alpha, ctype_alnum	Yes
typechk_num	type checking numeric function	is_int, is_double, is_float, is_integer is_long, is_numeric, is_real, is_scalar, ctype_digit	No
contentchk	content checking function	preg_match, preg_match_all, ereg, eregi strnatcmp, strcmp, strncmp, strncasecmp, strcasecmp	No
fillchk	fill checking function	isset, empty, is_null	Yes
cond	*if* instruction presence	if	No
join_str	join string function	implode, join	No
erase_str	erase string function	trim, ltrim, rtrim	Yes
replace_str	replace string function	preg_replace, preg_filter, str_ireplace, str_replace ereg_replace, eregi_replace, str_shuffle, chunk_split	No
split_str	split string function	str_split, preg_split, explode, split, spliti	Yes
add_str	add string function	str_pad	Yes/No
sub_str	substring function	substr	Yes/No
sub_str_replace	replace substring function	substr_replace	Yes/No
char5	substring with less than 6 chars	–	No
char6	substring with more than 5 chars	–	Yes
start_where	where the substring starts	–	Yes/No
conc	concatenation operator	–	Yes/No
var_vv	variable tainted	–	Yes
miss	miss value	–	Yes/No

Table 5.1: Intermediate Slice Language tokens.

`char6` token, assuming by default that that amount can carry malicious data, i.e., is tainted. The same scenario appears in the `contentchk` token that depends of the verification pattern.

Grammar

The ISL grammar is composed of the rules shown in Figure 5.1. It is used to translate a slice, composed of code elements (Table 5.1, column 3), into what we designate by *slice-isl*, composed of tokens (column 1). A *slice-isl* is the result of the application of a set of `statement` rules (line 2), each one of which can be a sub-rule (lines 4-11), an `if` statement (line 12) or an assignment instruction (line 13). The sub-rules represent the syntax of the functions in column 3 of the table: sensitive sink (line 4), sanitization (line 5), validation (line 6), extraction and modification (lines 7-10), and concatenation (line 11). Each rule denotes how each code element is represented, as exemplified above for the `mysql_query` function and its parameter, where the `sensitive_sink` rule was applied (line 4 on Figure 5.1).

```
1  grammar isl {
2    slice-isl: statement+
3    statement:
4        sensitive_sink: ss (param | concat)
5      | sanitization: sanit_f param
6      | valid: (typechk_str | typechk_num | fillchk | contentchk) param
7      | mod_all: (join_str | erase_str | replace_str | split_str) param
8      | mod_add: add_str param num_chars param
9      | mod_sub: sub_str param num_chars start_where?
10     | mod_rep: sub_str_replace param num_chars param start_where?
11     | concat: (statement | param) (conc concat)?
12     | cond statement+ cond?
13     | (statement | param) attrib_var
14   param: input | var
15   attrib_var: var
16   num_chars: char5 | char6
17 }
```

Listing 5.1: Grammar rules of ISL.

A HMM processes observations from left to right and a PHP assignment instruction assigns the right-hand side to the left-hand side; the assignment rule in ISL follows the HMM scheme. This means, for example, that the PHP instruction `$u = $_GET['user'];` is translated to `input var`, where `input` is the right-hand side and `var` the left one.

5.2.2 Variable map

A *slice-isl* does not contain information about the variables represented by the `var` token. However, this information is crucial for the vulnerability detection process as `var` may apply to different variables and the existence of a vulnerability may depend on that information. Therefore, during slice translation a data structure called *variable map* is populated. This map associates each occurrence of `var` in the *slice-isl* with the name of the variable that appears in the source code. This allows tracking how input data propagates to different variables or is sanitized/validated or modified. Each line of the variable map starts with 1 or 0, indicating if the instruction is an assignment or not. The rest of the line contains one item per token in a *slice-isl* instruction. For instance, the above PHP instruction, `$u = $_GET['user'];`, translated to `input var`, populates the variable map with the entrance `1 - u`, denoting that that instruction is an assignment containing a variable in the second position. The – symbolizes a place within of the instruction not occupied by a variable.

5.2.3 Slice translation process

The process of slice translation consists in representing the slice using ISL and creating the corresponding variable map. This section presents this process with two examples.

The slice extractor analyses the source code, extracting slices that start in entry points and end in sensitive sinks. The instructions between these points are those that handle entry points and variables depending on them. The slice extractor performs intra- and inter-procedural analysis, as it tracks the entry points and their dependencies along the source code, walking through different files and functions. The analysis is also context-sensitive as it takes into account the results of function calls.

Figure 5.2(a) shows PHP code (a slice) vulnerable to SQLI and Figure 5.2(b) shows this code translated into ISL and the corresponding variable map (ignore the right-hand side for now). The first line represents the assignment of an input to a var: `input var` in ISL. The variable map entry starts with 1 (assignment) and has two items, one for `input` (–) and the other for `var` (`u`, the variable name without the $ character). The next line is a variable assignment represented by `var var` in ISL and by `1 u q` in the variable map. The last line contains a sensitive sink (`ss`) and two variables.

The second example is in Figure 5.3. The slice extractor takes from that code two slices: lines {1, 2, 3} and {1, 2, 4}. The first has input validation, but not the second that is vulnera-

```
1 $u = $_POST['username'];
2 $q = "SELECT pass FROM users WHERE user='".$u."'";
3 $result = mysql_query($q);
```

(a) code with SQLI vulnerability.

	slice-isl	variable map	tainted list	slice-isl classification
1	input var	1 - u	TL = {u}	⟨input,Taint⟩ ⟨var_vv_u,Taint⟩
2	var var	1 u q	TL = {u, q}	⟨var_vv_u,Taint⟩ ⟨var_vv_q,Taint⟩
3	ss var var	1 - q result	TL = {u, q, result}	⟨ss,N-Taint⟩ ⟨var_vv_q,Taint⟩ ⟨var_vv_result,Taint⟩

(b) *slice-isl* (c) outputting the final classification

Figure 5.2: Code vulnerable to SQLI, translation into ISL, and detection of the vulnerability.

```
1 $u = $_POST['name'];
2 if (isset($u) && preg_match('/[a-zA-Z]+/', $u))
3     echo $u;
4 echo $u;
```

(a) code with XSS vulnerability and validation.

	slice-isl	variable map	list
1	input var	1 - u	TL = {u}; CTL = {}
2	cond fillchk var contentchk var cond	0 - - u - u -	TL = {u}; CTL = {u}
3	cond ss var	0 - - u	TL = {u}; CTL = {u}
4	ss var	0 - u	TL = {u}; CTL = {}

(b) *slice-isl* and variable map (c) artifacts lists

Figure 5.3: Code with a slice vulnerable to XSS (lines {1, 2, 4}) and a slice not vulnerable (lines {1, 2, 3}), with translation into ISL.

ble to XSS. The corresponding ISL and variable map are shown in the middle columns. The interesting cases are lines 2 and 3 that represent the `if` statement and its true branch. Both are prefixed with the `cond` token and the former also ends with the same token.

5.3 The Model

This section presents the *model* used to learn and detect vulnerabilities. The section covers the two phases of the proposed approach (Section 5.1). The learning phase is mainly presented in Sections 5.3.1 and 5.3.2 (parameters). The detection phase is presented in Section

5.3.3. In the learning phase, the corpus (a set of annotated sequences of observations) is used to set the *parameters* of the sequence model (matrices of probabilities). In the detection phase, a sequence of observations represented in ISL is processed by the model using the Viterbi algorithm (Jurafsky & Martin, 2008) with some adaptations to decode the sequence of states that explains those observations. This algorithm is often used in NLP to decode (i.e., discover) the states given the observations. The states classify the observations as tainted or not; and in particular the last state of the sequence indicates if the slice is vulnerable or not.

5.3.1 Building the corpus

Our approach involves configuring the model automatically using machine learning. The *corpus* is a set of sequences of observations annotated with states, that contains the knowledge that will be learned by the model. The corpus is crucial for the approach as it includes the information about which sequences of instructions lead to vulnerabilities or not.

The corpus is built in four steps: *collecting* a set of (PHP) instructions associated with slices vulnerable and not vulnerable; *representing* these instructions in ISL (sequences of observations); *annotating* manually the state to each observation (to each ISL token) of the sequences; and *removing* duplicated sequences of observations annotated with states. The upper part of Figure 5.1(a) represents these steps.

The most critical step is the first, in which a set of slices representing existing vulnerabilities (and non-vulnerabilities) with different combinations of code elements has to be obtained. In practice we used a large number of slices from open source applications (see Section 5.4).

A sequence of the corpus is composed of two or more pairs ⟨`token,state`⟩. The instruction `$var = $_POST['paramater'];`, for instance, translated into ISL becomes *`input var`* and is represented in the corpus as ⟨`input,Taint`⟩ ⟨`var_vv,Taint`⟩.
Both states are *`Taint`* (compromised) because the *`input`* is always *`Taint`* (*`input`* is the source of attacks we consider).

In the corpus, the sequences of observations are annotated according to their taintdness status and type, as presented in column 4 of Table 5.1, and the tokens representing some class of functions from that table. For instance, the PHP instruction `$var =`

`htlmentities($_POST['parameter'])` is translated to *sanit_f input var* and represented in the corpus by the sequence ⟨`sanit_f,San`⟩ ⟨`input, San`⟩ ⟨`var,N-Taint`⟩. The first two tokens were annotated with the *San* state, because the sanitization function sanitizes its parameter, and the last token was annotated with *N-Taint* state, meaning that the operation and the final state of the sequence are not tainted.

Notice that in the previous examples the state of the last observation is the final state of the sequence. In the sanitization example that state is *N-Taint*, indicating that the sequence is not-tainted (not compromised), while in the other example that state is *Taint*, indicating that the sequence is tainted (compromised).

As mentioned above, the token `var_vv` is not produced when slices are translated into ISL, but used in the corpus to represent variables with state *Taint* (tainted variables). In fact, during translation into ISL variables are not known to be tainted or not, so they are represented by the token `var`. In the corpus, if the state of the variable is annotated as *Taint*, the variable is represented by `var_vv`, forming the pair ⟨`var_vv,Taint`⟩.

Listings 5.2 and 5.3 show an example of this process of creating of the corpus, with its four steps. Listing 5.2(a) presents PHP instructions extracted from vulnerable and non-vulnerable slices. Two examples of these slices, respectively, are the sequences of instructions of the lines {1, 8} and {2, 5, 8}. Listing 5.2(b) represents each of these instructions into ISL (second step). Some instructions have more than one representation, depending if the extracted slice is vulnerable or not. For example, the instruction labeled by 5 has two representations (the two lines immediately below of it) to represent the sanitization of an untainted and a tainted variable, respectively (first and second representations). In the figure, it is visible the difference between the `var` and `var_vv` tokens. For the two examples of slices above, line 8 is represented in ISL by the first representation for the vulnerable slice, and by the second representation for the non-vulnerable slice. Listing 5.3 represents the last two steps and the corpus. Each sequence of observations is annotated as explained above. The duplicated sequences are reduced to one sequence, because different PHP instructions can result in the same sequence. For example, the PHP instructions from lines 1 and 2 (Listing 5.2(a)) result in the sequence of line 1 of the corpus.

```
1 $var = $_POST['parameter']
2 $var = $_GET['parameter']
3 $var = htmlentities($_POST['parameter'])
4 $var = mysql_real_escape_string($_GET['parameter'])
5 $var = htmlentities($var)
6 $var = "SELECT field FROM table WHERE field = $var"
7 $var = mysql_query($var)
8 echo $var
9 include($var)
10 if (isset($var) && $var > number)
11 if (is_string($var) && preg_match('pattern', $var))
```

(a) collecting step.

```
1 $var = $_POST['parameter']
      input var_vv
2 $var = $_GET['parameter']
      input var_vv
3 $var = htmlentities($_POST['parameter'])
      sanit_f input var
4 $var = mysql_real_escape_string($_GET['parameter'])
      sanit_f input var
5 $var = htmlentities($var)
      sanit_f var var
      sanit_f var_vv var
6 $var = "SELECT field FROM table WHERE field = $var"
      var var
      var_vv var_vv
7 $var = mysql_query($var)
      ss var var
      ss var_vv var_vv
8 echo $var
      ss var_vv
      ss var
9 include($var)
      ss var_vv
      ss var
10 if (isset($var) && $var > number)
      cond fillchk var_vv cond
      cond fillchk var cond
11 if (is_string($var) && preg_match('pattern', $var))
      cond typechk_str var_vv contentchk var_vv cond
      cond typechk_str var_vv contentchk var cond
      cond typechk_str var contentchk var_vv cond
      cond typechk_str var contentchk var cond
```

(b) representing step.

Listing 5.2: Building the corpus: collecting and representing steps.

```
<input,Taint> <var_vv,Taint>
<sanit_f,San> <input,San> <var,N-Taint>
<sanit_f,San> <var,San> <var,N-Taint>
<sanit_f,San> <var_vv,San> <var,N-Taint>
<var,N-Taint> <var,N-Taint>
<var_vv,Taint> <var_vv,Taint>
<ss,N-Taint> <var,N-Taint> <var,N-Taint>
<ss,N-Taint> <var_vv,Taint> <var_vv,Taint>
<ss,N-Taint> <var_vv,Taint>
<ss,N-Taint> <var,N-Taint>
<cond,N-Taint> <fillchk,Val> <var_vv,Val> <cond,N-Taint>
<cond,N-Taint> <fillchk,Val> <var,Val> <cond,N-Taint>
<cond,N-Taint> <typechk_str,Val> <var_vv,Val> <contentchk,Val>
    <var_vv,Val> <cond,N-Taint>
<cond,N-Taint> <typechk_str,Val> <var_vv,Val> <contentchk,Val> <var,Val>
    <cond,N-Taint>
<cond,N-Taint> <typechk_str,Val> <var,Val> <contentchk,Val> <var_vv,Val>
    <cond,N-Taint>
<cond,N-Taint> <typechk_str,Val> <var,Val> <contentchk,Val> <var,Val>
    <cond,N-Taint>
```

Listing 5.3: Building the corpus: annotating and removing steps.

5.3.2 Sequence model

Vocabulary and states

The HMM vocabulary consists in the 21 ISL tokens. The HMM contains the 5 states in Table 5.2. The final state of *slice-isl* will be vulnerable (`Taint`) or not vulnerable (`N-Taint`), but for correct detection it is necessary to take into account sanitization (`San`), validation (`Val`) and modification (`Chg_str`) of the user inputs. Therefore these three factors are represented as intermediate states in the model.

State	Description	Emitted observations
Taint	Tainted	input, var, var_vv, conc
N-Taint	Not tainted	Input, var, var_vv, ss, cond, conc
San	Sanitization	input, var, var_vv, sanit_f
Val	Validation	input, var, var_vv, typechk_str, typechk_num, contentchk, fillchk
Chg_str	Change string	input, var, var_vv, join_str, add_str, erase_str, replace_str, split_str, sub_str, sub_str_replace, char5, char6, start_where

Table 5.2: HMM states and the observations they emit.

Model graph

The model uses the knowledge in the corpus to discover the states of new sequences of observations, detecting vulnerabilities. The knowledge that we want to be learned can be expressed as a graph, which represents the model to detect vulnerabilities. Figure 5.4 shows the graph for the specific HMM we use, where the nodes represent the states and the edges the transitions between them. Table 5.2 shows the observations that can be emitted in each state (column 3).

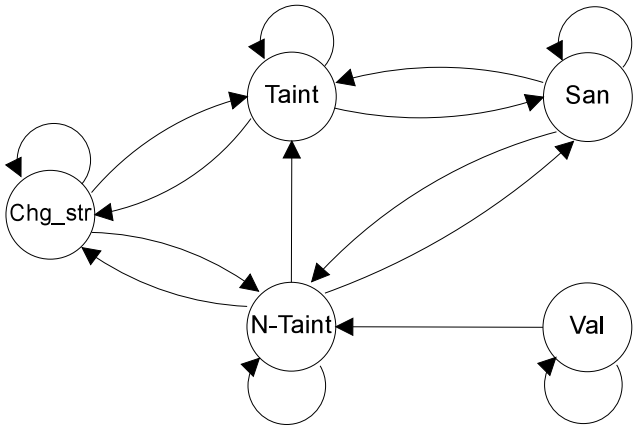
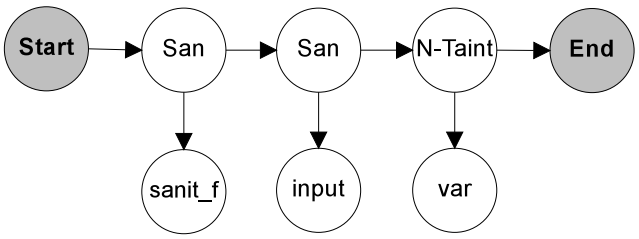


Figure 5.4: Model graph of the proposed HMM.

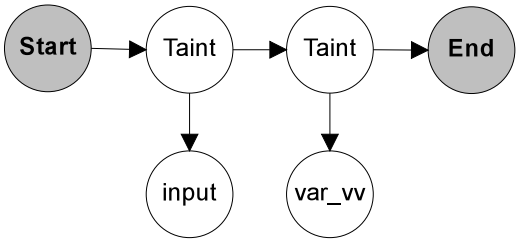
A sequence of observations can start in any state except `Val`, and end in the states `Taint` or `N-Taint`. The exception is due to validated instructions that begin with the `cond` observation (e.g., lines 2-3 in Figure **??**), which is emitted by the `N-Taint` state, but after this observation the state transits to the `Val` state. In relation to the final state, an instruction (a sequence of observations) from *slice-isl* is classified for all its observations, where the state of the last observation will be the final state of all observations, meaning that an instruction is always classified as `Taint` or `N-Taint`. Therefore, the final state of the last instruction of *slice-isl* gives the final classification, i.e., says if the *slice-isl* is vulnerable or not. State outputs and transitions depend on the previously processed observations and the knowledge learned.

Figure 5.5 shows the instantiation of the graph for two sequences. The sanitization code of Figure 5.5(a) is translated to the ISL sequence `sanit_f input var`. The sequence starts in the `San` state and emits the `sanit_f` observation; next it remains in the same

state and emits the *input* observation; then, it transits to *N-Taint* state, emitting the *var* observation (non-tainted variable). Figure 5.5(b) depicts the assignment of an entry point to a variable, turning this one tainted (*Taint*) and emitting *var_vv* (tainted variable).



(a) PHP instruction: `$p = htlmentities($_GET['user'])`
ISL instruction: `sanit_f input var`
Sequence: ⟨`sanit_f,San`⟩ ⟨`input,San`⟩ ⟨`var,N-Taint`⟩



(b) PHP instruction: `$u = $_GET['user']`
ISL instruction: `input var`
Sequence: ⟨`input,Taint`⟩ ⟨`var_vv,Taint`⟩

Figure 5.5: Models for two example corpus sequences.

Parameters

The *parameters* of the model are probabilities for the initial states, the state transitions, and symbol emissions (Section 5.3.2). The parameters are calculated using the corpus and the add-one smoothing technique to ensure that all probabilities are different from zero.

The probabilities are calculated from the corpus counting the number of occurrences of observations and/or states for each type of probability. The result are 3 matrices of probabilities with dimensions of $(1 \times s)$, $(s \times s)$ and $(t \times s)$, where s and t are the number of states and tokens of the model. For our model these numbers are 5 and 21, resulting in matrices of dimensions (1×5), (5×5) and (21×5). They are calculated as follows:

- *Initial-state probabilities:* count how many sequences start in each state. Then, calculate the probability for each state dividing these counts by the number of sequences of the corpus, resulting in a matrix with the dimension (1×5). For example, to obtain the initial-state probability to the `San` state, we need to count how many sequences begin with the `San` state; then this number is divided by the *corpus* size.

- *Transition probabilities:* count how many times in the corpus a certain state transits to another state (or to itself). Recall that we consider pairs of states. We can calculate the transition probability by dividing this count by the number of pairs of states from the corpus that begin with the start state. For instance, the transition probability from the `N-Taint` state to `Taint` state is the number of occurrences of this pair of states divided by the number of pairs of states starting in the `N-Taint` state. The resulting matrix has a dimension of (5×5), that represents the possible transitions between the 5 states.

- *Emission probabilities:* count how many times in the corpus a certain token is emitted by a certain state, i.e., count how many times a certain pair $\langle$`token,state`$\rangle$ appears in the corpus. Then, calculate the emission probability by dividing this count by the total of pairs $\langle$`token,state`$\rangle$ for that specific state. An example is the probability of the `Taint` state to emit the `var_vv` token – the pair $\langle$`var_vv,Taint`$\rangle$. First, the number of occurrences of this pair in the corpus is counted, next it is divided by the total of pairs related to the `Taint` state. The resulting matrix – called *global emission probabilities matrix* – has a dimension of (21×5), representing the 21 tokens emitted by the 5 states.

Zero-probabilities have to be avoided because the Viterbi algorithm uses multiplication to calculate the probability of the next state, and therefore we need to ensure that this multiplication is never zero. The *add-one smoothing* technique (Jurafsky & Martin, 2008) is used to calculate the parameters, avoiding zero probabilities. This technique adds a unit to all counts, making zero-counts equal to one and the associated probability different from zero. For example, to calculate the probability of the state `Vul` being emitted the observation `ss`, means to count how many times the pair $\langle$`ss,Vul`$\rangle$ appears in the *corpus*; if it is equal to zero, then the result is a zero-probability. Using this technique, this probability is transformed in a non-zero-probability.

5.3.3 Detecting vulnerabilities

This section describes the *detection* phase of Figure 5.1(b).

Detection

A sequence of observations in ISL is processed by the model using the Viterbi algorithm to decode the sequence of states. For each observation, the algorithm calculates the probability of each state emitting that observation, taking for this purpose the emission and transition probabilities and the maximum of probabilities calculated for the previous observation in each state, i.e., the order in which the observation appears in the sequence and the previous knowledge. For the first observation of the sequence the initial-state probabilities are used, whereas for the rest of the probabilities these are replaced by the maximum of probabilities calculated for each state for the previous observation. For emission probabilities, the matrix for the observations to be processed is retrieved from the global emission probabilities matrix. The multiplication of these probabilities is calculated for each state – *score of state* – and the maximum of scores is selected, assigning it the state with bigger score to the observation. The process is repeated for all observations and the last observation is the one with the highest probability of the states of the sequence. In our case, this probability classifies the sequence as `Taint` or `N-Taint`.

A *slice-isl* is composed by a set of sequences of observations. The model is applied to each sequence, classifying each one as tainted or not (`Taint`, `N-Taint`). However, for the classification to be correct the model needs to know which variables are tainted and propagate this information between the sequences processed. For this purpose, three artefacts are used in the model: the lists of tainted variables (*tainted list*, TL) (explained next), inputs and tainted variables validated by validation functions (*conditional tainted list*, CTL), and sanitized variables (*sanitized list*, SL) (Section 5.3.3).

There are two relevant interactions between the *variable map*, the emission probabilities and `var_vv` to fill the three lists in two moments of the sequence processing: *after* and *before*.

- *After:* if the sequence represents an assignment, i.e., the last observation of the sequence is a `var`, the variable map is visited to get the variable name for that `var`, then

TL is updated: (i) inserting the variable name if the state is `Taint`; or (ii) removing it if its state is `N-Taint` and the variable belongs to TL. In case (ii) and in the presence of a sanitization sequence, SL is updated inserting the variable name; if the sequence represents an `if` condition (the first and last observations of the sequence must be `cond`), for each `var` and `var_vv` observation, the variable map is visited to get the variable name, next TL to verify if it contains the variable name, and then, in that case, CTL is updated inserting that variable name.

- *Before:* for each `var` observation, the variable map is visited to get the variable name, then TL and SL are accessed to verify if they contain that variable name. CTL is also accessed if the sequence starts with the token `cond`; in case of variable name only belong to TL, the `var` observation is updated to `var_vv`, then the emission probabilities matrix for the observations from the sequence is retrieved from the global emission probabilities matrix.

In order to detect vulnerabilities, the Viterbi algorithm was modified with these artefacts and interactions. Our model processes each sequence of observations from *slice-isl* as follows:

1. *"before"* is performed.
2. the decoding step of the Viterbi algorithm is applied.
3. *"after"* is performed.

Detection example

Figure 5.2 shows an example of detection. The figure contains from left to right: the code, the *slice-isl*, the variable map, and TL after the model classifies the sequence of observations. Observing TL, it is visible that it contains the tainted variables and that they propagate their state to the next sequences, influencing the emission probability of the variable. In line 1, the `var` observation is vulnerable because by default the `input` observation is so; the model classifies it correctly; and in TL the variable `u` is inserted. Next, line 2, before the Viterbi algorithm is applied the first `var` observation is updated to `var_vv` because it represents the `u` variable which belongs to TL. The `var_vv var` sequence is classified by the Viterbi

algorithm, resulting in `Taint` as final state, and the variable `q` is inserted in TL. The process is repeated in the next line.

Figure 5.2(c) presents the decoding of *slice-isl*, where it is possible to observe the replacement of `var` by `var_vv`, with the variable name as suffix. Also, the states of each observation are presented and the state of the last observation indicates the final classification (there is a vulnerability). Looking for the states generated it is possible to understand the execution of the code without running it, why the code is vulnerable, and which variables are tainted.

Validation and sanitization

The *conditional tainted list* (CTL) is an artefact used to help interpret inputs and variables that are validated. This list will contain the *validated inputs* and *variables*, i.e., the inputs (token `input`) and tainted variables that belong to TL, and that are validated by validation functions (tokens `typechk_num` and `contentchk`). Therefore, when line 2 of Figure 5.3 is processed, this list is created and will be passed to the other sequences. That figure contains two *slice-isl* executed alternatively, depending on the result of the condition in line 2: {1, 2, 3} and {1, 2, 4}. When the model processes the former, it sets TL = {u} and CTL = {u}, as the variable {u} is the parameter of the `contentchk` token. The final state of the *slice-isl* (corresponding to line 3) is `N-Taint`, as the variable is in CTL. In the other slice there is no interaction with CTL and the final state is `Taint`.

The *sanitized list* (SL) is a third artefact. Its purpose is essentially the same as CTL, except that SL will contain variables sanitized using sanitization functions or modified using functions that, e.g., manipulate strings.

5.4 Implementation and Assessment

To evaluate our approach and model we implemented them in the DEKANT tool. Moreover, we defined a corpus that we used to train the model before running the experiments. This corpus can be later extended with additional knowledge (remember that the tool is able to learn, so also to evolve).

5.4.1 Implementation of the DEKANT

The DEKANT tool was implemented in Java. The tool has four main modules: *knowledge extractor*, *slice extractor*, *slice translator*, and *vulnerability detector*.

Knowledge extractor

The *knowledge extractor* module is independent of the other three and executed just when the corpus is first created or later modified. It runs in three steps.

1. *Corpus processing:* the sequences of the corpus are loaded from a plain text file; each sequence is separated in pairs ⟨`token,state`⟩ and the elements of each pair are inserted in the matrices of *observations* and *states*.

2. *Parameter calculation:* the parameters (probabilities) of the model are computed using the two matrices, and inserted in auxiliary matrices.

3. *Parameter storage:* the parameters are stored in a plain text file to be loaded by the *vulnerability detector* module.

To obtain the parameters we need to *normalize the sequence length of the corpus*, making it equal for all sentences. Processing the sequences of the corpus (corpus processing step) means splitting the observations from the states, resulting the *observations* and *states* matrices with equal dimension. The number of columns of these matrices represents the sequence length. However, the sequences of the corpus do not have the same length (see Figure 5.3, for example), so normalization is necessary.

The model (and tool) uses the token `miss` for this purpose. The model is configured with a maximum length sequence (e.g., 10), which it is automatically calculated when the sequences of the corpus are processed, finding which of them has the largest length. The sequences with length lower than the maximum are padded to their right with the pair ⟨`miss,Taint`⟩ or ⟨`miss,N-Taint`⟩, depending on the state of the last element of the sequence. Recall that the last state of the sequence indicates its final state, so it can be used for padding without causing wrong classification.

Slice extractor

The *slice extractor* extracts slices from PHP code by tracking data flows starting at entry points and ending at sensitive sinks, independently if the entry points are sanitized, validated and modified.

Slice translator

The *slice translator* parses the slices, translates them into ISL applying the grammar, and generates the variable maps.

Vulnerability detector

The *vulnerability detector* works in three steps.

1. *Parameter loading*: the parameters (probabilities) are loaded from a text file and stored in matrices (initial-state, transition and emission illustrated in Figure 5.6 extracted from the corpus presented in next section).

2. *Sequence of observations decoding*: the modified Viterbi algorithm is executed, i.e., the process described in Section 5.3.3 is performed.

3. *Evaluation of sequences of observations*: the probability of a sequence of observations to be explained by a sequence of states is estimated, the most probable is chosen, and a vulnerability flagged if it exists.

In step 2., if the length of the sequence being processed is bigger than the configured maximum sequence length (retrieved from the corpus), the sequence is divided in sequences of that maximum sequence length, and each one is classified separately, but the initial probability from the next sequence is equal to the resulting probability from the previous sequence.

5.4.2 Model and corpus assessment

A concern when specifying a HMM is to make it accurate and precise, i.e., to ensure that it classifies correctly sequences of observations or, in our case, that it detects vulnerabilities correctly. *Accuracy* measures the total of slices well-classified as vulnerable and non-vulnerable, whereas *precision* measures the fraction of vulnerabilities identified that are really vulnerabilities. The objective is high accuracy and precision or, equivalently, minimum

rates of false positives (inexistent vulnerabilities classified as vulnerabilities) and false negatives (vulnerabilities not classified as vulnerabilities). The model is configured with the corpus, so its accuracy and precision depend strongly on that corpus containing correct and enough information.

We created a corpus with 510 slices: 414 vulnerable and 96 non-vulnerable. These slices were extracted from several open source PHP applications[1] and contained vulnerabilities from the eight classes presented in Section 2.1 (SQLI, XSS, RFI, LFI, DT/PT, SCD, OSCI and PHPCI). The knowledge extracted from this corpus is shown in Figure 5.6, representing the parameters of the model.

$$\begin{bmatrix} 0.062 & 0.323 & 0.062 & 0.015 & 0.538 \end{bmatrix}$$

(a) initial-state probabilities.

$$\begin{bmatrix} 0.619 & 0.099 & 0.174 & 0.059 & 0.333 \\ 0.115 & 0.641 & 0.304 & 0.353 & 0.373 \\ 0.027 & 0.028 & 0.435 & 0.059 & 0.020 \\ 0.009 & 0.033 & 0.043 & 0.471 & 0.020 \\ 0.009 & 0.006 & 0.043 & 0.059 & 0.255 \end{bmatrix}$$

(b) transition probabilities.

$$\begin{bmatrix} 0.085 & 0.015 & 0.103 & 0.030 & 0.075 \\ 0.016 & 0.294 & 0.051 & 0.212 & 0.075 \\ 0.326 & 0.010 & 0.154 & 0.030 & 0.075 \\ 0.008 & 0.005 & 0.256 & 0.030 & 0.015 \\ 0.008 & 0.051 & 0.026 & 0.030 & 0.015 \\ 0.380 & 0.406 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.091 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.091 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.061 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.061 & 0.015 \\ 0.008 & 0.076 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.134 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.104 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.134 \\ 0.008 & 0.061 & 0.026 & 0.030 & 0.015 \\ 0.070 & 0.020 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.015 \\ 0.270 & 0.208 & 0.056 & 0.061 & 0.015 \end{bmatrix}$$

(c) global emission probabilities.

Figure 5.6: Parameters of the model extracted from the corpus. The columns represent the 5 states in the order that appears in the first column of Table 5.2. The lines of matrix (c) are the tokens in the order appearing in the first column of Table 5.1

[1] bayar, bayaran, ButterFly, CurrentCost, DVWA 1.0.7, emoncms, glfusion-1.3.0, hotelmis, Measureit 1.14, Mfm-0.13, mongodb-master, Multilidae 2.3.5, openkb.0.0.2, Participants-database-1.5.4.8, phpbttrkplus-2.2, SAMATE, superlinks, vicnum15, ZiPEC 0.32, Wordpress 3.9.1.

To evaluate the accuracy and precision of the model configured with this corpus, we did *10-fold cross validation* (Demšar, 2006), a common technique to validate training data. This form of validation involves dividing the training data (the corpus of 510 slices) in 10 folds. Then, the tool is trained with a pseudo-corpus of 9 of the folds and tested with the 10th fold. This process is repeated 10 times to test every fold with the model trained with the rest. This estimator allows assessing the quality of the corpus without the bias of testing data used for training or just a subset of the data.

		Observed	
		Vulnerable	Not Vulnerable
Predicted	Vulnerable	412	16
	Not Vulnerable	2	80

Table 5.3: Confusion matrix of the model tested with the corpus. *Observed* is the reality (414 vulnerable slices, 96 not vulnerable). *Predicted* is the output of DEKANT with our corpus (428 vulnerable, 82 not vulnerable).

The confusion matrix of Table 5.3 presents the results of this estimator. The precision and accuracy of the model were around 96%. The rate of false positives was 17% and the rate of false negatives almost null (0.5%). There is a tradeoff between these two rates and it is better to have a very low rate of false negatives that leads to some false positives (non-vulnerabilities flagged as vulnerabilities) than the contrary (missing vulnerabilities). These results show that the model has good performance using this corpus.

5.5 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions using DEKANT and the corpus presented in the previous section:

1. Is *a tool that learns to detect vulnerabilities* able to detect vulnerabilities in plugins and real web applications? (Section 5.5.1)

2. Can it be more accurate and precise than other tools that do data mining using standard classifiers? (Section 5.5.2)

3. Can it be more accurate and precise than other tools that do data mining using standard classifiers? (Section 5.5.3)

4. Is it able to classify correctly vulnerabilities independently of their class? (Section 5.5.1) Is it able to classify correctly vulnerabilities independently of their class? (Section 5.5.1)

5.5.1 Open source software evaluation

To demonstrate the ability of DEKANT to classify vulnerabilities, we run it with 10 WordPress plugins (WordPress, 2015) and 10 packages of real web applications, all written in PHP, using the corpus of the previous section. *The code used in the evaluation was not the same used to build the corpus.*

Zero-day vulnerabilities in plugins

WordPress is the most adopted CMS worldwide and supports plugins developed by different teams. Plugins are interesting because they are often less scrutinized than full applications. We selected 10 plugins based on two criteria: development team and number of downloads. For the former, we choose 5 plugins developed by companies and the other 5 by individual developers. For the second, we choose 5 with less than 1000 downloads and the other 5 with more than 21,000 downloads. The plugins with less downloads were not always those developed by individual developers.

WordPress has a set of functions that sanitize and validate different data types, which are used by some of the plugins. Therefore, to run DEKANT with the source code of the plugins but without the WordPress code base, we added the information about those functions to the tool. Notice that the entry points and sensitive sinks remain mostly the same, except for sinks that handle SQL commands (*$wpdb* class). We configured DEKANT with these functions, mapping them to the ISL tokens. Recall that ISL abstracts the PHP instructions, so it can capture behaviors such as sanitization and validation even for the functions that were added.

DEKANT discovered 16 new vulnerabilities as shown in Table 5.4. 80 slices were extracted and translated into ISL. The tool classified 24 slices as vulnerable and 56 as not vulnerable (N-Vul), but 8 of the vulnerable were false positives (FP). This classification was confirmed by us manually. The 16 real vulnerabilities detected (columns 3-5) were 6 SQLI, 8 XSS, and 2 DT/LFI. These vulnerabilities were reported to the developers, who confirmed and fixed them, releasing new versions. The plugins *appointment-booking-calendar 1.1.7*,

easy2map 1.2.9, *payment-form-for-paypal-pro 1.0.1*, *resads 1.0.1* and *simple-support-ticket-system 1.2* were fixed thanks to this work. We registered the vulnerabilities in CVE with the IDs shown in the table.

Plugin	Slices	Real vulnerabilities			N-Vul	FP
		SQLI	XSS	DT & LFI		
appointment-booking-calendar 1.1.7*	12	1	3	–	6	2
		CVE-2015-7319, CVE-2015-7320				
calculated-fields-form 1.0.60	3	–	–	–	2	1
contact-form-generator 2.0.1	5	–	–	–	4	1
easy2map 1.2.9*	6	–	1	2	3	0
		CVE-2015-7668, CVE-2015-7669				
event-calendar-wp 1.0.0	6	–	–	–	6	0
payment-form-for-paypal-pro 1.0.1*	11	–	2	–	8	1
		CVE-2015-7666				
resads 1.0.1*	2	–	2	–	0	0
		CVE-2015-7667				
simple-support-ticket-system 1.2*	20	5	–	–	15	0
		CVE-2015-7670				
wordfence 6.0.17	6	–	–	–	6	0
wp-widget-master 1.2	9	–	–	–	6	3
Total	80	6	8	2	56	8

*confirmed and fixed by the developers and registered in CVE

Table 5.4: Vulnerabilities found by DEKANT in WordPress plugins.

The 16 zero-day vulnerabilities were found in 5 plugins: 2 developed by companies and 3 by individual developers; plus 2 having more than 21,000 downloads. These results show that, independently of the development teams and the number of downloads, the WordPress plugins are vulnerable and may contain more vulnerabilities than other web applications, as recent research suggests (Nunes *et al.*, 2015).

Real web applications

To demonstrate the ability of DEKANT to classify vulnerabilities from the 8 classes of Section 2.1, we run it with 10 open source software packages with vulnerabilities disclosed in the past. These packages were not used to build the corpus.

DEKANT classified 310 slices of the 10 applications. The results are in Table 5.5, columns 10-13. After this process we confirmed this classification manually in order to assess the results of DEKANT and the other tools (columns 2-5; Vul stands for vulnerable, San for sanitized, and VC for validated and/or changed). The 4 right-hand columns of

Web application	Slices				WAP				DEKANT			
	Vul	San	VC	Total	Vul	FPP	FP	FN	Vul	N-Vul	FP	FN
cacti-0.8.8b	2	0	8	10	2	2	6	0	2	6	2	0
communityEdition	16	36	8	60	16	6	2	0	16	44	0	0
epesi-1.6.0-20140710	25	1	8	34	25	6	2	0	25	5	4	0
NeoBill0.9-alpha	19	0	0	19	19	0	0	0	19	0	0	0
phpMyAdmin-4.2.6-en	1	6	7	14	1	0	7	0	1	13	0	0
refbase-0.9.6	5	4	3	12	5	0	3	0	5	1	6	0
Schoolmate-1.5.4	120	0	0	120	117	0	0	3	120	0	0	0
VideosTube	1	0	2	3	1	1	1	0	1	2	0	0
Webchess 1.0	20	0	0	20	18	0	0	2	20	0	0	0
Zero-CMS.1.0	2	5	11	18	2	5	6	0	2	16	0	0
Total	211	52	47	310	206	20	27	5	211	87	12	0

Table 5.5: Results of running the slice extractor, WAP and DEKANT in open source software.

the table show that DEKANT correctly classified 211 slices as being vulnerable (Vul) and the remaining as not-vulnerable (N-Vul), except 12 wrongly classified as vulnerable (false positives – FP). This misclassification is justified by the presence of validation and string modification functions (e.g., *preg_match* and *preg_replace*) with context-sensitive states. In such cases we set DEKANT to classify the slices as vulnerable but printing a warning on a possible false positive. Table 5.6 shows the confusion matrix summarizing these values. Overall, DEKANT had accuracy and precision of 96% and 95%, 12% of false positives, and no false negatives.

	Observed							
	DEKANT		WAP		Original		Analyzed	
Predicted	Vul	N-Vul	Vul	N-Vul	Vul	N-Vul	Vul	N-Vul
Vul	211	12	206	27	182	36	50	218
N-Vul	0	87	5	72	86	821	109	748

Table 5.6: Confusion matrix of DEKANT, WAP and C4.5/J48 in PhpMinerII data set (original and analyzed).

Table 5.7 summarizes the results and presents additional metrics. For the 10 packages, more than 4,200 files and 1,525,865 lines of code were analyzed and 223 vulnerabilities found (12 false positives). The largest packages were *epesi* and *phpMyAdmin* (741 and 241 thousand lines of code).

Table 5.8 presents the 223 slices classified by DEKANT as vulnerable (12 false positives) distributed by the 6 classes of vulnerabilities. Interestingly, all false positives were PHPCI

Web application	Files	Lines of code	Analysis time (s)	Vuln. files	Vulner. found
cacti-0.8.8b	249	95,274	7	7	4
communityEdition	228	217,195	21	11	16
epesi-1.6.0-20140710	2246	741,440	90	13	29
NeoBill0.9-alpha	620	100,139	5	5	19
phpMyAdmin-4.2.6-en	538	241,505	12	1	1
refbase-0.9.6	171	109,600	8	5	11
Schoolmate-1.5.4	64	8,411	2	41	120
VideosTube	39	3,458	2	1	1
Webchess 1.0	37	7,704	2	5	20
Zero-CMS.1.0	21	1,139	2	2	2
Total	4,213	1,525,865	151	91	223

Table 5.7: Summary of results of DEKANT with open source code.

and XSS vulnerabilities. The tool correctly classified the sanitized slices as not vulnerable. The vulnerabilities correctly classified by DEKANT correspond to 21 entries of vulnerabilities that appear in CVE (CVE, 2015) and OSVDB (OSVDB, 2015), as shows the Table 5.9.

Web application	SQLI	RFI, LFI DT/PT	PHPCI	XSS	Total
cacti-0.8.8b	0	0	2	2	4
communityEdition	4	4	3	5	16
epesi-1.6.0-20140710	0	3	4	22	29
NeoBill0.9-alpha	0	2	0	17	19
phpMyAdmin-4.2.6-en	0	0	0	1	1
refbase-0.9.6	0	0	0	11	11
Schoolmate-1.5.4	69	0	0	51	120
VideosTube	0	0	0	1	1
Webchess 1.0	6	0	0	14	20
Zero-CMS.1.0	1	0	0	1	2
Total	80	9	9	125	223

Table 5.8: Results of the classification of DEKANT considering different classes of vulnerabilities extracted by the slice extractor.

5.5.2 Comparison with data mining tools

To answer the second question, DEKANT was compared with WAP (version 2.1 presented in Section 3) and PHPMinerII with the 10 packages of the previous section. We opted by

Web application	Vulnerability
cacti-0.8.8b	XSS: CVE-2014-5026, CVE-2014-4082
communityEdition	LFI: CVE-2014-8770
	SQLI: CVE-2013-4580
	XSS: CVE-2014-2016, CVE-2013-5913
	PHPCI: CVE-2014-2988
epesi-1.6.0-20140710	XSS: OSVDB-103888
NeoBill0.9-alpha	XSS: OSVDB-86204
	DT, LFI: OSVDB-100669, OSVDB-100670
phpMyAdmin-4.2.6-en	XSS: CVE-2014-4955
refbase-0.9.6	XSS: OSVDB-44977, OSVDB-58139, CVE-2008-6400
Schoolmate-1.5.4	XSS: CVE-2010-5010
	SQLI: CVE-2010-5011
VideosTube	XSS: OSVDB-114753
Webchess 1.0	SQLI, XSS: Bugtraq 43895
Zero-CMS.1.0	XSS: CVE-2014-4195
	SQLI: CVE-2014-4034

Table 5.9: Registered vulnerabilities detected by DEKANT.

evaluating these tools with those packages and not with the plugins, because they are not configurable for the plugins. When run with the plugins these tools provide much worse results than DEKANT.

Both tools also classify slices previously extracted, but using data mining based on standard classifiers, which do not consider order. WAP performs taint analysis to extract the slices that start in an entry point and reach a sensitive sink, with attention to sanitization, then uses data mining to predict if they are false positives or real vulnerabilities. The tool deals with the same vulnerability classes as DEKANT. PhpMinerII uses data mining to classify slices as being vulnerable or not, without considering false positives. This tool handles only SQLI and reflected XSS vulnerabilities.

Comparison for all vulnerability classes

Columns 6 to 9 of Table 5.5 present WAP's results for the 8 vulnerability classes. WAP reported 206 vulnerabilities (Vul), 20 false positives predicted (FPP), with 27 false positives and 5 false negatives (vulnerabilities not detected). WAP identified the same 258 slices without sanitization (columns 2 and 4 from Table 5.5) than the slice extractor and detected the same 206 vulnerabilities than DEKANT (5 less than DEKANT, false negatives, FN). Moreover and as expected, from the 47 slices classified as not vulnerable by DEKANT,

WAP predicted correctly 20 of them as false positives (FPP), meaning that 27 slices were wrongly classified as vulnerabilities (FP), reporting 27 false positives.

This difference of false positives is justified by: (1) the presence of symptoms in the slice which are not contemplated by WAP as attributes in its data set; (2) lack of verification of the relations between attributes, once the data mining mechanism only verifies the presence of the attributes in the slice, does not relates them. The false negatives are justified by reason (2) plus the importance of the order of the code elements in the slice. The misclassification was based in the concatenation of variables tainted with not-tainted (variables validated or modified), in that order; then data mining matches the presence of symptoms related with validation and classified the slices as false positives. In these 5 slices is evident the importance of the order of code elements for a correct classification and detection. DEKANT implements a sequence model that takes into account that order, prevailing in these cases.

Columns 4 and 5 of Table 5.6 present the confusion matrix with these values. WAP had an accuracy of 90%, a precision of 88%, 2% of false negatives and 27% of false positives (Table 5.11, third column).

Comparison for SQLI and reflected XSS

For a fair comparison with PHPMinerII, only SQLI and reflected XSS vulnerabilities classes considered. Table 5.10 shows the results; columns 2 to 4 are the 158 vulnerabilities classified by DEKANT (80 SQLI, 78 XSS) and 6 false positives. The next four columns are about WAP, with the 153 vulnerabilities (77 SQLI, 76 XSS), but with 21 false positives and 5 false negatives. The next 12 columns present the PHPMinerII results.

Web application	DEKANT			WAP				PhpMinerII - SQLI						PhpMinerII - XSS						Pixy			
	SQLI	XSS	FP	SQLI	XSS	FP	FN	Yes	No	Y - Y	Y - N	FP	FN	Yes	No	Y - Y	Y - N	FP	FN	SQLI	XSS	FP	FN
cacti-0.8.8b	0	2	0	0	2	6	0	0	0	0	0	0	0	6	11	2	0	4	0	0	6	4	0
communityEdition	4	5	0	4	5	0	0	5	0	0	0	5	4	43	521	0	0	43	5	5	8	13	9
epesi-1.6.0-20140710	0	18	0	0	18	0	0	0	0	0	0	0	0	1	1	1	0	0	16	0	1	0	17
NeoBill0.9-alpha	0	17	0	0	17	0	0	0	0	0	0	0	0	20	3	17	0	3	0	0	20	3	0
phpMyAdmin-4.2.6-en	0	1	0	0	1	6	0	0	0	0	0	0	0	24	74	0	0	24	1	–	25	24	0
refbase-0.9.6	0	5	6	0	5	3	0	3	0	0	0	3	0	82	115	0	1	82	5	3	93	96	5
Schoolmate-1.5.4	69	14	0	66	14	0	3	41	11	11	0	30	58	2	0	2	0	0	12	303	113	339	6
VideosTube	0	1	0	0	1	0	0	10	19	0	0	10	0	2	28	1	0	1	0	12	2	13	0
Webchess 1.0	6	14	0	6	12	0	2	1	0	1	0	0	5	13	7	13	0	0	1	92	206	279	1
Zero-CMS.1.0	1	1	0	1	1	6	0	6	2	1	1	5	0	9	65	1	0	8	0	6	7	11	0
Total	80	78	6	77	76	21	5	66	32	13	1	53	67	202	825	37	1	165	40	421	481	782	38

Table 5.10: Comparison of results between DEKANT, WAP, PHPMinerII and Pixy with open source projects.

PhpMinerII does not come trained, so we had to create a data set to train it. For that purpose, PhpMinerII extracts slices that end in a sensitive sink, but that do not have to start

in an entry point. It outputs the slices, the vector of attributes of each slice, and a preliminary classification as vulnerable or not. Then a classification has to be assigned to each attribute vector manually. This data set is used to train the data mining part of the tool. We present experimental results of the tool running it both without and with data mining. Table 5.10 shows the analysis without data mining and the intersection of both sets of slices for SQLI and XSS with the DEKANT slices. For these two classes, columns 9, 10, 15 and 16 (Yes, No) show the number of slices classified by the tool, columns 11 and 17 (Y - Y) show the intersection (the number of vulnerabilities detected by both tools), whereas columns 12 and 18 (Y - N) depict the number of vulnerabilities that DEKANT classified correctly but PHPMinerII did not report. We observe that from the SQLI vulnerabilities detected by DEKANT, PHPMinerII only detected correctly approximately 16%, presenting high rates of false negatives and false positives. For XSS, PHPMiner II presents again an elevated rate of false negatives and false positives, besides a small number of true positives compared with the number of vulnerabilities detected by DEKANT.

To perform the data mining process the WEKA tool was used (Witten *et al.*, 2011) with the same classifiers as PhpMinerII (Shar & Tan, 2012b,c). The best classifier was the C4.5/J48. Columns 6 to 9 of Table 5.6 show the results of this classifier. The first two columns of these four are relative to the slices flagged by the tool without data mining, while the last two columns are relative to the data mining process presented above. The accuracy and precision are equal to 71% and 19%, and the false positives and negatives rates are 23% and 69%, justifying the very low precision rate.

Metric	**DEKANT**	**WAP**	**PhpMinerII**		**Pixy**
			original	**analyzed**	
acurracy	96%	90%	89%	71%	21%
precision	95%	88%	83%	19%	16%
false positive	12%	27%	4%	23%	84%
false negative	0%	2%	32%	69%	23%

Table 5.11: Evaluation metrics of DEKANT, WAP, PhpMinerII, Pixy.

Table 5.11 summarizes the comparison between DEKANT, WAP and PhpMinerII. DEKANT was the best of all. WAP was the second, also with low false negatives but high false positives. Despite PhpMinerII presenting the lowest false positive rate, it had the highest rate of false negatives and lower accuracy and precision rates, making it the weakest tool (false negatives are specially problematic as they represent vulnerabilities that were not found).

5.5.3 Comparison with taint analysis tools

We compare DEKANT with Pixy (Jovanovic *et al.*, 2006), a tool that performs taint analysis to detect SQLI and reflected XSS vulnerabilities, taking sanitization functions in consideration. The last four columns of Table 5.10 are related to the analysis made with Pixy. Despite Pixy reporting 902 vulnerabilities in 10 packages, they are mostly false positives. Those vulnerabilities were 421 SQLI and 481 XSS (first two columns of the last 4). The same process of the previous section was executed over the results of Pixy. In summary, only 120 vulnerabilities are the same as for DEKANT, while the rest are false positives and some false negatives (last 2 columns).

5.6 Discussion

DEKANT is a static analysis tool because it searches for vulnerabilities in source code, without execution. DEKANT has two main parts: one programmed, another learned. The former corresponds to the slice extractor that does part of what other static analysis tools do: parses the code and extracts slices. The latter uses the sequence model we propose, configured with knowledge extracted from the corpus.

In classic static analysis tools this knowledge was programmed, involving several data structures and variables representing and relating the code elements that create and avoid vulnerabilities. Programming this knowledge is a hard, complex task, for the programmers, who may leave errors that lead to false positives and false negatives (Dahse & Holz, 2015).

Taking this difficulty into account, machine learning started to be used to reduce the effort required to programming static analysis tools. Table 5.11 compares the results of WAP and PHPMinerII (both use machine learning) with Pixy (an older tool that does not use it). In that table it is possible to see that tools based on machine learning can provide good results. The application of data mining requires a definition of a data set with the knowledge about vulnerabilities, making it a crucial part of the process for correct detection.

WAP does taint analysis and alias analysis for detecting vulnerabilities, although it goes further by also correcting the code. Furthermore, Pixy does only module-level analysis, whereas WAP does global analysis (i.e., the analysis is not limited to a module or file, but can involve several). We propose an alternative approach that does not involve coding knowledge about vulnerabilities, instead based on training a model through annotated code samples.

The slices extracted from the source code, i.e excerpts of code that begin in a entry points ans end in a sensitive sink, are processed by DEKANT mechanism to discover if they are vulnerabilities or not.

Our work also aims to identify the location of vulnerabilities in source code, contrarily to other works that assess the quality of the software in terms of the prevalence of defects (Arisholm *et al.*, 2010; Briand *et al.*, 2000; Lessmann *et al.*, 2008) and vulnerabilities (Neuhaus *et al.*, 2007; Perl *et al.*, 2015; Shar & Tan, 2012b,c; Shin *et al.*, 2011; Walden *et al.*, 2009) (details about these works in Section 2.3). WAP is quite different because it has to identify the location of vulnerabilities in the source code, so that it can correct them with fixes. Moreover, WAP does not use data mining to identify vulnerabilities, but to predict whether the vulnerabilities found by taint analysis are really vulnerabilities or false positives.

This chapter presents the first static analysis approach and tool that learns to detect vulnerabilities automatically using machine learning (WAP has most knowledge programmed and PHPMinerII does not identify vulnerabilities, only predicts if they exist). Furthermore, we go one step further by using for the first time in this context a sequence model instead of standard classifiers. This model not only considers the code elements that appear in the slices, but also their order and relations between them. Again, similarly to what happens with standard classifiers, the definition of the corpus for the sequence model is crucial. Table 5.11 compares the results of DEKANT with WAP and PHPMinerII, showing that this approach indeed improves the results.

5.7 Conclusions

The chapter explores a new approach to detect web application vulnerabilities inspired in NLP in which static analysis tools *learn* to detect vulnerabilities automatically using machine learning. Whereas in classical static analysis tools it is necessary to code knowledge about how each vulnerability is detected, our approach obtains knowledge about vulnerabilities automatically. The approach uses a sequence model (HMM) that, first, learns to characterize vulnerabilities from a corpus composed of sequences of observations annotated as vulnerable or not, then processes new sequences of observations based on this knowledge, taking into consideration the order in which the observations appear. The model can be used as a static analysis tool to discover vulnerabilities in source code and identify their location.

Future developments may consider the usage of other types of sequence classification models (e.g., discriminative approaches such as Conditional Random Fields or Structured-SVMs, or even recently proposed methods based on deep neural network architectures), which often had lead to better results in the context of NLP tasks, and which also facilitate the inclusion of additional contextual features.

The DEKANT tool implements the proposed approach. It was experimented with 10 packages of open source PHP applications and 10 WordPress plugins. 16 zero-day vulnerabilities were found in the analyzed plugins. They were confirmed and fixed by the developers and registered in CVE by us. These plugins were fixed due to this work. This evaluation suggests that the tool can detect vulnerabilities from several classes, having an accuracy of around 96% and performing better that other tools in the literature.

6

Preventing Injection Attacks inside the DBMS

After more than a decade of research, web application security continues to be a challenge and the backend database the most appetizing target. For example, SQL injection (SQLI) attacks have allegedly victimized 12 million Drupal sites (BBC Technology, 2014); SQLI attacks were considered an important threat against critical infrastructures (ICS-CERT, 2015); and stored cross-site scripting (XSS) attacks were used to inject malicious code in servers running Wordpress (Search Security TechTarget, 2015).

The mechanisms most commonly used to protect web applications from malicious inputs are web application firewalls (WAFs), sanitization/validation functions, and prepared statements in the application source code. The first two mechanisms, respectively, inspect web application inputs and block and sanitize those that are considered malicious/dangerous, whereas the third bounds inputs to placeholders in the query. Other anti-SQLI mechanisms have been presented in the literature, but barely adopted. Some of these mechanisms monitor SQL queries and block them if they deviate from certain query models. However, they can make mistakes because the queries are inspected without full knowledge about how the server-side scripting language and the DBMS process them (Boyd & Keromytis, 2004; Buehrer *et al.*, 2005; Halfond & Orso, 2005; Masri & Sleiman, 2015; Su & Wassermann, 2006).

In all these cases, administrators and programmers make assumptions about how the server-side language and the DBMS work and interact, which sometimes are simplistic,

others blatantly wrong. For example, programmers often assume that in PHP the function `mysql_real_escape_string` always effectively sanitizes inputs and prevents SQLI attacks but unfortunately this is not true. In addition, they may ignore that data may be unsanitized when inserted in the DBMS leading to second-order SQLI vulnerabilities.

We argue that such simplistic or wrong assumptions are caused by a *semantic mismatch* between how an SQL query is expected to run and what actually occurs when it is executed. This mismatch leads to unexpected vulnerabilities in the sense that mechanisms such as those mentioned above can become ineffective, resulting in false negatives (attacks not detected). To avoid this problem, these attacks could be handled after the server-side code processes the inputs and the DBMS validates the queries, reducing the amount of assumptions that are made. The mismatch and this solution are not restricted to web applications.

Today operating systems are much more secure than years ago due to the deployment of automatic protection mechanisms in themselves, in core libraries (e.g., .NET and glibc), and in compilers. For example, address space layout randomization (ASLR), data execution prevention (DEP), or canaries/stack cookies are widely deployed in Windows and Linux (Howard & LeBlanc, 2007; Koschany, 2013). These mechanisms block a large range of attacks irrespectively of the programmer following secure programming practices or not. Clearly, something similar would be desirable for web applications. The DBMS is an interesting location to add these protections as it is a common target for attacks.

We propose modifying – "hacking" – DBMSs to detect and block attacks in runtime without programmer intervention. We call this approach *SElf-Protecting daTabases preventIng attaCks* (SEPTIC). This chapter, focus on the two main categories of attacks related with databases: *SQL injection* attacks, which continue to be among those with highest risk (Williams & Wichers, 2013) and for which new variants continue to appear (Ray & Ligatti, 2012), and *stored injection attacks*, which also involve SQL queries. For SQLI, we propose detecting attacks essentially by comparing queries with query models, taking to its full potential an idea that has been previously used only outside of the DBMS (Boyd & Keromytis, 2004; Buehrer *et al.*, 2005; Halfond & Orso, 2005; Su & Wassermann, 2006) and circumventing the semantic mismatch problem. For stored injection, we propose having plugins to deal with specific attacks before data is inserted in the database.

We demonstrate the concept with a popular deployment scenario: MySQL, probably the most popular open-source DBMS (DB-Engines, 2015), and PHP, the language most used in

web applications (more than 82%) (Imperva, 2015). We also explore Java/Spring, the second most employed programming language.

The chapter is organized as follows. Section 6.1 presents the injection attacks we consider in the chapter. In Section 6.2 the SEPTIC approach is presented, starting by an overview, then detailing the components that compose the SEPTIC mechanism and how it is trained to detect injection attacks. Section 6.3 describes the implementation of SEPTIC in the MySQL DBMS and the interaction with the PHP Zend and Spring/Java engines to collect query identifiers. Section 6.4 presents an evaluation of the performance overhead of SEPTIC. Next, Section 6.5 presents how the mechanism can be extended to other DBMSs and used to identify vulnerabilities and protect non-web applications. Finally, the chapter ends with a discussion of the related work and conclusions in Section 6.6.

6.1 DBMS Injection Attacks

We define *semantic mismatch* as the difference between how programmers assume SQL queries are run by the DBMS and how queries are effectively executed. This mismatch often leads to mistakes in the implementation of protections in the source code of web applications, letting these applications vulnerable to SQL injection and other attacks involving the DBMS. The semantic mismatch is subjective in the sense that it depends on the programmer, but some mistakes are usual. A common way to try to prevent SQLI consists in sanitizing user inputs before they are used in SQL queries. For instance, PHP `mysql_real_escape_string` precedes special characters like the prime or the double prime with a backslash, transforming these delimiters into normal characters. However, sanitization functions do not behave as envisioned when the special characters are represented differently from expected. This problem has lead us to use the term semantic mismatch to refer to the gap between how the SQL queries that take these sanitized inputs are believed to be executed by the programmer, and how they are actually processed by the DBMS.

We identified several DBMS injection attacks in the literature, including a variety of cases related to semantic mismatch (Clarke, 2009; Douglen, 2007; Dowd *et al.*, 2006; Ray & Ligatti, 2012; Son *et al.*, 2013). Table 6.1 organizes these attacks in classes. The first

three columns identify the classes, whereas the fourth and fifth state what PHP sanitization functions and the DBMS do to the example malicious inputs in the sixth column.

Class		Class name	PHP sanit. func.	DBMS	Example malicious input
SQL injection	**A**	Obfuscation			
	A.1	- Encoded characters	do nothing	decodes and executes	`%27, 0x027`
	A.2	- Unicode characters	do nothing	translates and executes	`U+0027, U+02BC`
	A.3	- Dynamic SQL	do nothing	completes and executes	`char(39)`
	A.4	- Space character evasion	do nothing	removes and executes	`char(39)/**/OR/**/1=1--`
	A.5	- Numeric fields	do nothing	interprets and executes	`0 OR 1=1--`
	B	Stored procedures	sanitize	executes	`admin' OR 1=1`
	C	Blind SQLI	sanitize	executes	`admin' OR 1=1`
Stored inj.	**D**	Stored injection code			
	D.1	- Second order SQLI	–	executes	any of the above
	D.2	- Stored XSS	–	–	`<script>alert('XSS')</script>`
	D.3	- Stored RCI, RFI, LFI	–	–	`malicious.php`
	D.4	- Stored OSCI	–	–	`; cat /etc/passwd`
	S.1	Syntax structure	sanitize	executes	`admin' OR 1=1`
	S.2	Syntax mimicry	sanitize	executes	`admin' AND 1=1--`

RCI: Remote Code Injection; RFI:Remote File Inclusion; LFI: Local File Inclusion; OSCI: OS Command Injection

Table 6.1: Classes of attacks against DBMSs

As mentioned in the introduction, we consider two main classes of attacks: *SQL injection* and *stored injection* (first column). These classes are divided in sub-classes for common designations of attacks targeted at DBMSs (A to D). Obfuscation attacks (class A) are the most obvious cases of semantic mismatch. Classes S.1 and S.2 classify attacks in terms of the way they affect the syntactic structure of the SQL query. Class S.1 is composed of attacks that modify this structure. Class S.2 is composed of attacks that modify the query but mimic its original structure.

```
1 $u = mysql_real_escape_string($_POST['username']);
2 $p = mysql_real_escape_string($_POST['password']);
3 $query = "SELECT * FROM users WHERE username='$u' AND password='$p'";
4 $result = mysql_query($query);
```

Listing 6.1: Script vulnerable to SQLI with encoded characters.

Class A, obfuscation, contains five subclasses. Consider the code excerpt in Listing 6.1 that shows a login script that checks if the credentials the user provides (username, password) exist in the database.[1] The user inputs are sanitized by the `mysql_real_escape_string` function (lines 1-2) before they are inserted in the query (line 3) and submitted to the DBMS (line 4). If an attacker injects the `admin'--` string as username (line 1), the `$user` variable

[1] All examples included in the chapter were tested with Apache 2.2.15, PHP 5.5.9 and MySQL 5.7.4

receives this string sanitized, with the prime character preceded by a backslash. The user `admin\'--` does not exist in the database so this SQLI attack is not successful.

On the contrary, this sanitization is ineffective if the input uses URL encoding (Berners-Lee *et al.*, 2005), leading to an attack of class A.1. Suppose the attacker inserts the same username URL-encoded: `%61%64%6D%69%6E%27%2D%2D%20`. `mysql_real_escape_string` function does not sanitize the input because it does not recognize `%27` as a prime. However, MySQL receives that string as part of a query and decodes it, so the query executed is `SELECT * FROM users WHERE username='admin'-- ' AND password='foo'`. The attack is therefore effective because this query is equivalent to `SELECT * FROM users WHERE username='admin'` (no password has to be provided as `--` indicates that the ret of the code in the line should be ignored). This is also an attack of class S.1 as the structure of the query is modified (the part that checks the password disappears). The other subclasses of class A involve similar techniques. In class A.2 the attacker encodes some characters in Unicode, e.g., the prime as `U+02BC`. In A.3 decoding involves calling dynamically a function (e.g., the prime is encoded as `char(39)`). Class A.4 attacks use spaces and equivalent strings to manipulate queries (e.g., concealing a space with a comment like `/**/`) (Clarke, 2009). A.5 attacks abuse the fact that numeric fields do not require values to be enclosed with primes, so a tautology similar to the example we gave for A.1 can be caused without these characters, fooling sanitization functions like `mysql_real_escape_string`.

Stored procedures that take user inputs may be exploited similarly to queries constructed in the application code (class B). These inputs may modify or mimic the syntactic structure of the query, leading to attacks of classes S.1 or S.2.

Blind SQLI attacks (class C) aim to extract information from the database by observing how the application responds to different inputs. These attacks may also fall in classes S.1 or S.2.

Class D attacks – stored injection – are characterized by being executed in two steps: the first involves doing an SQL query that inserts attacker data in the database (`INSERT`, `UPDATE`); the second uses this data to complete the attack. The specific attack depends on the data inserted in the database and how it is used in the second step. In a second order SQLI attack (class D.1) the data inserted is a string specially crafted to be inserted in a second SQL query executed in the second step. This second query is the attack itself, which may fall in classes S.1 or S.2. This is another case of semantic mismatch as the sanitization created by functions like `mysql_real_escape_string` is removed by the DBMS when the string is

inserted in the database (first step of the attack). A stored XSS (class D.2) involves inserting a browser script (typically JavaScript) in the database in the first step, then returning it to one or more users in the second step. In class D.3 the data inserted in the database can be a malicious PHP script or an URL of a website containing such a script, resulting on local or remote file inclusion, or on remote code injection. In class D.4 attacks the data that is inserted is an operating system command, which is executed in the second step.

6.2 The SEPTIC Approach

This section presents the SEPTIC approach. The idea consists in having a module inside the DBMS that processes every query it receives in order to detect attacks against the DBMS. We designate both the approach and this module by SEPTIC. This approach circumvents the semantic mismatch problem as detection is performed near the end of the data flow entering the DBMS, just before is executed the query.

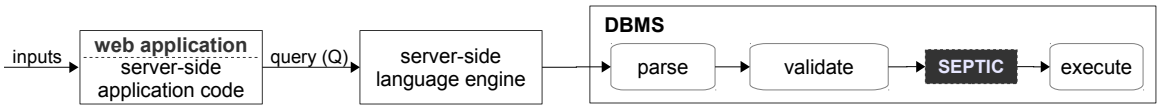
6.2.1 SEPTIC overview

This section presents an overview of the approach. Figure 6.1(a) represents the architecture of a web application, including the DBMS and SEPTIC. This module is placed inside the DBMS, after the parsing and validation of the queries. There may be also hooks inside the server-side language engine (Section 6.2.3).

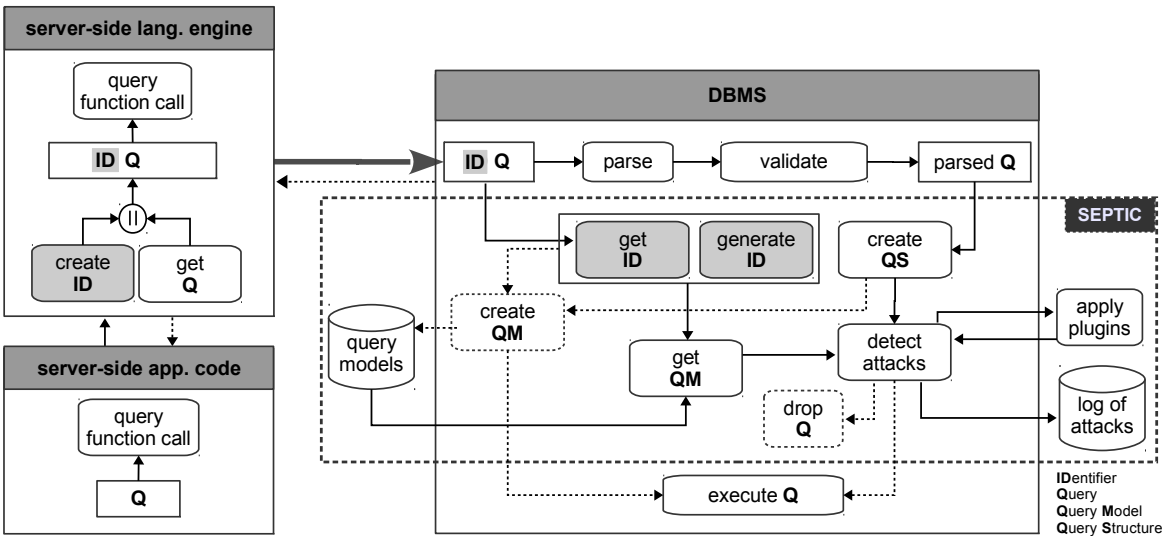
In runtime SEPTIC works basically the following way:

1. *Server-side application code:* requests the execution of a query Q;
2. *Server-side language engine:* receives Q and sends it to the DBMS; optionally it may add an identifier (ID) to Q;
3. *DBMS:* receives, parses, validates, and executes Q; between validation and execution, SEPTIC detects and possibly blocks an incoming attack.

Figure 6.1(b) provides more details on the operation of SEPTIC. The figure should be read starting from the gray arrow at the top/left. Dashed arrows and dashed processes represent alternative paths.



(a) Main modules of a web application backed by a DBMS with SEPTIC.



(b) SEPTIC approach data flows.

Figure 6.1: Architecture and data flows of a web application and SEPTIC (optional components in gray).

When a web application is started, SEPTIC has to undergo some training before it enters in normal execution. Training is typically done by putting SEPTIC in *training mode* and running the application for some time without attacks (Section 6.2.5). Training results in a set of query models (QM) stored in SEPTIC.

In normal execution, for every query SEPTIC receives, it extracts the query ID and the query structure (QS). If no ID is provided, SEPTIC generates one (Section 6.2.3). SEPTIC detects attacks first by comparing the query structure (QS) with the query model(s) stored for that ID. If there is no match, an SQLI attack was detected. Otherwise, SEPTIC uses a set of *plugins* to discover stored injection attacks. If no attack is detected the query is executed.

The action taken when an attack is detected depends on the mode SEPTIC is running. In *prevention mode*, SEPTIC aborts the attacks, i.e., it drops the queries and the DBMS stops the query processing. In *detection mode*, queries are executed, not dropped. In both modes of operation, SEPTIC logs information about the attacks detected.

In summary, SEPTIC runs in three modes, one for training (*training mode*) and two for normal operation (*prevention mode* and *detection mode*).

The following sections present the approach in detail.

6.2.2 Query structures and query models

As explained in the previous section, in prevention and detection modes SEPTIC finds out if a query is an attack by comparing the *query structure* with the *query model(s)* associated to the query's ID.

We consider that SEPTIC receives the parse tree of every query represented as a *list of stacks* data structure. Each stack of the list represents a clause of the query (e.g., `SELECT`, `FROM`, `WHERE`), and each of its nodes contains data about the query element, such as category (e.g., field, function, operator), data type (e.g., integer, string), and data.

The *query structure* (QS) of a query is constructed by creating a single stack with the content of all the stacks in the list of stacks of a query. Figure 6.2 depicts a generic query structure, showing from bottom to top the clauses and their elements. Each *node* (a row) represents an element of the query.

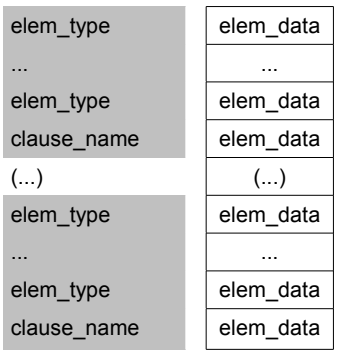
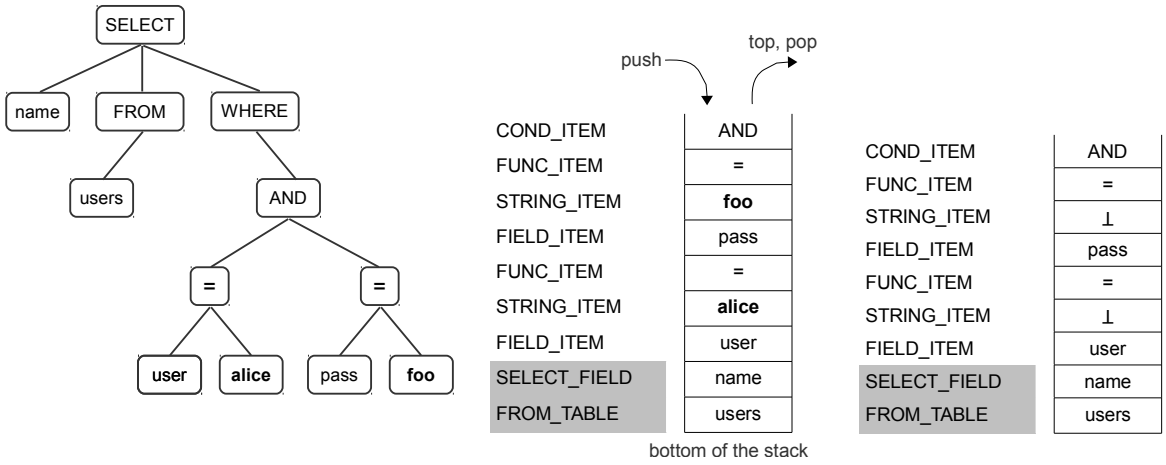


Figure 6.2: A generic query structure.

Each node is composed by the element type (category) and the element data: ⟨ELEM_TYPE, ELEM_DATA⟩. The single exception is the alternative format ⟨DATA_TYPE, DATA⟩ that represents an input inserted in the query and its (primitive) data type (DATA_TYPE). A part of the query is considered to be an input if its type is primitive (e.g., a string or an integer) or if it

is compared to something in a predicate. For the clauses with conditional expressions (e.g., `WHERE`) the elements are inserted in the QS by doing post-order traversal of the parse tree of the query (i.e., the left child is visited and inserted in the stack first, then the right child, and so on until the root).



(a) Query parse tree (b) Query structure (c) Query model

Figure 6.3: Representation of a query as parse tree, structure (QS) and model (QM).

As mentioned in the previous section, in training mode SEPTIC creates query models. Specifically, it creates a *query model* (QM) whenever the DBMS processes a query, but stores it only if that model is not yet stored for that query ID. The query model is created based on the query structure of the query. The process consists simply in substituting DATA by a special value ⊥ in all ⟨DATA_TYPE, DATA⟩ nodes to denote that these fields shall not be compared during attack detection (Section 6.2.4). All the other nodes are identical in QM and QS.

Take as example the query `SELECT name FROM users WHERE user='alice' AND pass='foo'`. Figure 6.3 represents its (a) parse tree, (b) structure (QS), and (c) model (QM). In Figure 6.3(b) and (c) the gray items at the bottom have data about the `SELECT` and `FROM` clauses, whereas the rest are about the `WHERE` clause. In Figure 6.3(b) the inputs are represented in bold and in Figure 6.3(c) they have the special value ⊥ as explained. In the left-hand column, each item of the query takes a category (field, data type, condition operator, etc), whereas the right-hand column has the query's keywords, variables and primitive data

type. Primitive data types (real, integer, decimal and string) also take a category, such as STRING_ITEM (e.g., in the third row).

6.2.3 Query identifiers

Each query received by the DBMS has to be verified against one or more query models. *Query identifiers* (IDs) are used to match queries to their models. More specifically, each query is assigned an ID and for each ID the training mode creates a set of one or more models. Then, during the detection/prevention mode, the SEPTIC module matches a query to a set of models. From the point of view of the module, IDs are opaque, i.e., their structure is not relevant.

SEPTIC can use three kinds of IDs, depending on where they are generated: in the server-side language engine (SSLE), in the DBMS, and outside both the SSLE and the DBMS. We explain them below.

SSLE-generated IDs

The SSLE is arguably the best place to generate the IDs because this can let the application administrator oblivious to the existence of IDs. Figure 6.1(b) shows how this would work generically (SSLE in the left-hand side).

Ideally, every query issued by an application should have a unique ID (Section 6.2.4) and the SSLE can provide this in many cases. For instance, in the example of Listing 6.1 there should be a unique ID for the query constructed in line 3 and issued in line 4. In training mode a model with this ID would be constructed and in prevention/detection modes any query issued there would have the same ID. This would allow queries to be compared against the model without confusion with queries issued elsewhere in the application source code. The SSLE can create this ID when it sees a call to function `mysql_query`. The ID may contain data such as the file and line number in which the query is issued. However, this may be not enough to distinguish the queries because many applications have a single function that calls the DBMS with different queries. This function is called from several places in the application, but the file and line number that calls the DBMS is always the same.

We consider the ID format to be a sequence of *file:line* pairs separated by the character |, one pair per each function entered while the query is being composed. Specifically, the first

pair corresponds to the line where the DBMS is called and the rest to lines where the query is passed as argument to some function. *file* contains the complete path to allow distinguishing even queries from different applications using the same DBMS.

Assume that the code sample of Figure 6.1 is in file `login.php`. The query is created in the same function that calls the function `mysql_query`, so the ID is simply *login.php:4*, meaning that the DBMS is called in line 4 of file `login.php`. Consider a second example in which line 4 is substituted by `$result = my_db_query($query)`, that function `my_db_query` is defined in file `my_db.php`, and that function calls the DBMS using `mysql_query` in line 10 of that file. In this example, the ID is *my_db.php:10 | login.php:4*. This ID format is not guaranteed to generate unique IDs in all situations, but we observed no cases in which it did not. In these examples we show the filename without the full pathname for readability.

DBMS-generated IDs

Whenever the SEPTIC module in the DBMS receives a query without ID (e.g., because the SSLE does not generate SEPTIC IDs), it generates an ID automatically (Figure 6.1(b), gray boxes inside SEPTIC). The DBMS is unaware of what kind of client calls it (e.g., if it is an SSLE), much less about the web application source code. Therefore this ID has a different format. Similarly to SSLE-generated IDs, the application administrator can be oblivious to DBMS-generated IDs.

The ID format is the SQL command (typically `SELECT`) followed by the number of nodes of the query structure. For the example of Listing 6.1 that has the query structure of Figure 6.3(b) the ID would be *select_9*.

IDs generated outside the DBMS and the SSLE

In the previous two kinds of IDs the web application administrator is left aside from the process of assigning IDs to queries. If for some reason these kinds of IDs are not desirable, the administrator can define his own IDs. These IDs can have any format, e.g., a sequential number or the same format used in SSLE-generated IDs. They can be added to the queries in a few ways: (1) they may be appended to the query when it is defined or when the DBMS is called; or (2) a wrapper may be inserted between the applications code and the DBMS.

6.2.4 Attack detection

This section explains how SEPTIC detects attacks by dividing the categories of Table 6.1 in two groups that are discovered differently: SQL injection and stored injection. The former contains the classes A to C and D.1, whereas the latter contains class D (except D.1). Class D.1 is also a form of stored injection, but it is more convenient to detect these attacks using the approach to discover SQLI.

SQLI detection

SEPTIC detects SQLI attacks by verifying if queries fall in classes S.1 and S.2. We say that attack classes S.1 and S.2 are *primordial for SQL injection* because any SQLI attack falls in one of these two categories. The rationale is that if an SQLI attack neither modifies the query structure (class S.1) nor modifies the query mimicking the structure (class S.2), then it must leave the query unmodified, but then it is not an SQL injection attack.

SEPTIC detects SQLI attacks by comparing each query with the query models for the query's ID *structurally* (for class S.1) and *syntactically* (for class S.2). An attack is flagged if there are differences between the query and all the models for its ID.

Given a query Q with a certain ID and its query structure QS, detection involves iterating over all the models QM_i stored for ID. For every QM_i there are two steps:

1. *Structural verification* – if the number of items in QS is different from the number of items in QM_i, then Q does not match the model QM_i and detection for QM_i ends;

2. *Syntactical verification* – if the data type of any of the items of QS is different from the type of any of the items of QM_i (except primitive types), then Q does not match the model QM_i and detection for QM_i ends. Items are compared starting at the top and going down the QS and QM stacks as represented in Figures 6.3(b) and (c). Primitive data types (real, integer, decimal and string) are an exception because DBMSs implicitly make type-casting between them (e.g., integer to string), so these types are considered equivalent.

This process is iterated for all query models QM_i stored for ID. If Q matches one of the models, there is no attack; otherwise there is an attack. The action taken depends on the mode in which SEPTIC is running: the query processing is aborted in prevention mode, and the query is executed in detection mode.

As mentioned in Section 6.2.3, IDs should be unique, so that a single query model QM would be stored for each ID during training. From that point of view *DBMS-generated IDs* are the worst option as they do not ensure uniqueness, except in applications with a very small number of queries. *SSLE-generated IDs* tend to be unique and *IDs generated outside the DBMS and the SSLE* may be created unique.

Stored injection detection

Stored injection attacks have two steps. In the first, malicious data is inserted in the database; in the second that data is taken from the database and used. For example, for stored XSS (D.2) the data includes a script to be executed at the victims' browsers; in the first step it is stored in the database; in the second step that script is taken from the database and sent to a browser. These attacks cannot be detected in the way just explained because they do not work by modifying queries. Therefore, we employ a different solution based on the idea of detecting the presence of malicious data.

SEPTIC detects the presence of malicious data in queries that insert data in the database (first step of the attacks). To do this detection, SEPTIC contains a set of *plugins*, typically one for each type of attack. The plugins analyze the queries searching for code that might be executed by browsers (JavaScript, VB Script), by an operating system (shell script, OS commands, binary files) or by server-side scripts (php). Since running the plugins may introduce some overhead, the mechanism is applied in two steps:

1. *Filtering* – searches for suspicious strings such as: <, >, `href`, and *javacsript* attributes (D.2); protocol keywords (e.g., `http`) and extensions of executable or script files (e.g., *exe, php*) (D.3); special characters (e.g., *;* and |) (D.4). If none is found, detection ends.

2. *Testing* – consists in passing the input to the proper plugin for inspection. For example, if the filtering phase finds the `href` string, the data is provided to a plugin that detects stored XSS attacks. This plugin inserts the input in a simple HTML page with the three main tags (`<html>`, `<head>`, `<body>`), and then calls an HTML parser to determine if other tags appear in the page indicating the presence of a script.

6.2.5 Training

As explained in Section 6.2.1, whenever an application is put to run, SEPTIC has to be subjected to training. This is necessary for SEPTIC to create the models of the queries for SQLI detection (Section 6.2.4). There are two methods to do training: *training phase* and *incremental*.

- *training phase* – involves putting SEPTIC in training mode and executing all queries of the web application with correct inputs (i.e., inputs that are not attacks). For every query a model is created and stored, unless the same model has already been stored for the same ID. If there is already a model (or more) associated to that ID and the model created is different, then ID becomes associated to two (or more) models. After this is done, SEPTIC can be put in prevention or detection mode and no further intervention from the administrator is needed. The execution of all queries can be achieved in two fashions: (1) using the unit tests of the application; or (2) with the assistance of an external module, called *septic_training*. This module is a web client that works as a crawler. For each web page, it searches for HTML forms and extracts information about the submission method, action, variables and values. Then, it issues HTTP requests for all forms, causing the SQL queries to be sent to the DBMS. These queries can contain user inputs generated by the training module, can be static, or can depend on the results of other queries.

- *incremental* – in this method, SEPTIC runs in prevention or detection mode all the time, without the need to switch modes and run an explicit training phase. This is very convenient and efficient as long as no attacks happen before the models are created. In both modes, for every query SEPTIC obtains the query structure (QS), gets the set of QMs associated to the query ID, and compares QS with every QM in the set, as explained in Section 6.2.4. From the point of view of training, the relevant case is when there is no QM associated with the ID. In this situation, SEPTIC behaves as if it was in the training phase and creates and stores the query model. The administrator is notified and should confirm that the model was built with a correct query, as it did not appear previously. This verification, however, is not critical for two reasons: (1) it is highly unlikely that the first query with a certain ID in a web application is malicious

(attackers take time to find the application and to learn how it works); (2) in the unlikely case of the model is built with a malicious query, this will become conspicuous as correct queries will start being detected as attacks, which will be conspicuous.

In case there are modifications to the application code we envisage two cases. If the changes are not significant, SEPTIC can continue in detection or prevention mode, building new QMs incrementally (incremental method). If the application code suffers many changes, SEPTIC can be put in training mode (training phase method) and all QMs of the application are rebuilt. In this case, the existent QMs are substituted by new ones. However, in both cases the administrator can opt for either training method.

An interesting case occurs if a query changes from line x to line y in the new version of application. This is not problematic if the training phase method is used, as all QMs are rebuilt. In the incremental method two unlikely scenarios may happen: (1) the QM of the query of line y is created and associated to a IDy not in use or to an existing IDy; (2) the IDx (query from line x) receives a new QM, if the line x has now a different query. In both cases the old QMs stored for IDx and IDy are checked for the queries that come with those IDs with the new version of the application. Even if SEPTIC checks that they not match with the old QMs, they match with the new QMs, so SEPTIC does not flag an attack (no false positives). False negatives (attacks not detected) are possible as a wrong QM will be associated to an ID, but this should not happen for two reasons: the two scenarios above are unlikely as a query would have to move to the same line of another; an attack against one of the queries would have to match exactly the QM of the other query.

6.2.6 Detection examples

This section presents two detection examples to illustrate the process.

SQLI detection

Consider a query `SELECT name FROM users WHERE user=? AND pass=?`. This query checks if a user exists in the database, returning his name. It accepts two inputs represented by a question mark. The corresponding query model is represented in Figure 6.3(c). Consider a second-order SQLI attack (class D.1): (1) a malicious user provides an input that

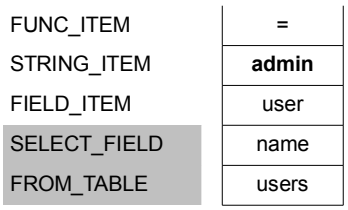
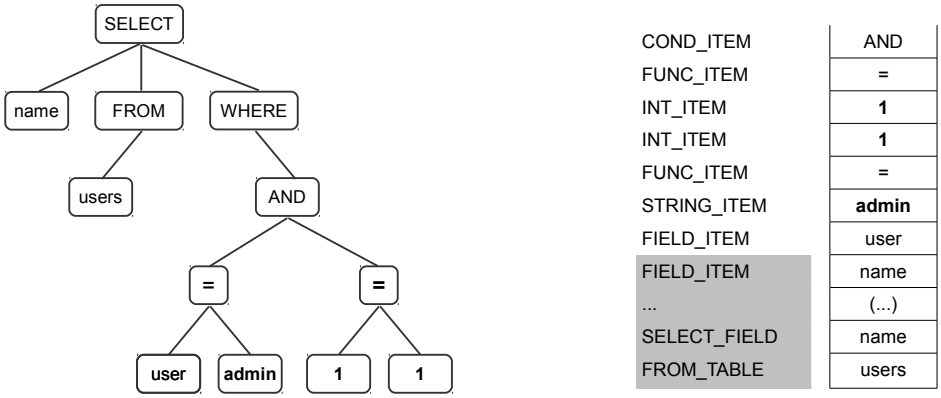


Figure 6.4: QS of query `SELECT name FROM users WHERE user=? AND pass=?` with `admin'--` as user.

leads the application to insert `adminU+02BC--` in the database (i.e., `admin'--` with the prime represented in unicode as `U+02BC`); (2) later this data is retrieved from the database and inserted in the `user` field in the query above; (3) the DBMS parses and validates the query, decoding `U+02BC` into the prime; the resulting query `SELECT name FROM users WHERE user= admin` falls in class S.1 as it modifies the structure of the query. Figure 6.4 presents the QS for this query and Figure 6.3(c) its QM, which we assume was stored in SEPTIC's *query models* store during training. When the query is issued, SEPTIC compares QS with QM and during structural verification observes that they do not match, as the number of items of both structures is different.

For a second example, consider a syntax mimicry attack (class S.2), the query above and the malicious input `admin' AND 1=1--` inserted as `user`. The resulting query is `SELECT name FROM users WHERE user= admin AND 1=1`. Figure 6.5 represents the parse tree and query stack of this query. Comparing the parse tree and stack with Figure 6.3(a) and (b), both trees have the same structure and both stacks equal number of nodes. When the query is issued, SEPTIC compares QS with QM. First, during structural verification it checks that they match, as the number of items of both structures is equal; then during syntactical verification it observes that the ⟨INT_ITEM, 1⟩ node from QS (fourth row in Figure 6.5(b)) does not match with the ⟨FIELD_ITEM, PASS⟩ node from QM (Figure 6.3(c)). The attack is flagged due to this difference.



(a) Query parse tree (b) Query stack

Figure 6.5: Stack of query with the *admin' AND 1=1* input.

Stored XSS detection

Consider a web application that registers new users and that a malicious user inserts as his first name `<script> alert("Hello!");</script>`, which is JavaScript code and a class D.2 attack. When SEPTIC receives the query, it does the filtering step and finds two characters associated with XSS, < and >, so it calls the plugin that detects stored XSS attacks. This plugin inserts this input in a web page, calls an HTML parser, finds that the input contains a script, and flags a stored XSS attack.

6.2.7 Discussion

To summarize, SEPTIC has the following important features:

- *Server-side language independence* – SEPTIC requires minimal and optional support at SSLE level to obtain the identifiers (unlike (Halfond *et al.*, 2008; Pietraszek & Berghe, 2005; Son *et al.*, 2013; Xu *et al.*, 2005));
- *No client configuration* – the DBMS client connectors do not need reconfiguration to use SEPTIC, as it is inside the DBMS;
- *Client diversity* – several DBMS clients of different types may be connected to a single DBMS server with SEPTIC;
- *No application source code modification* – the programmer does not need to make changes to the web application source code to use the mechanism (unlike (Boyd &

Keromytis, 2004; Buehrer *et al.*, 2005; Halfond & Orso, 2005; Su & Wassermann, 2006; Xu *et al.*, 2005));

- *No application source code analysis* – SEPTIC does not need to do source code analysis to find the queries in the source code of the web application (unlike (Bandhakavi *et al.*, 2007; Halfond & Orso, 2005));

- *Vulnerability diagnosis* – SEPTIC can use the query identifiers to identify the place where the vulnerability exists in the source code when an attack is detected (unlike (Boyd & Keromytis, 2004; Buehrer *et al.*, 2005; Halfond & Orso, 2005; Halfond *et al.*, 2008; Pietraszek & Berghe, 2005; Son *et al.*, 2013; Su & Wassermann, 2006; Trustwave SpiderLabs, 2015; Xu *et al.*, 2005)); details in Section 6.5.2.

6.3 Implementation

This section explains how SEPTIC was implemented in MySQL and the creation of identifiers implemented in two contexts: for PHP applications by modifying the PHP runtime (Zend engine); and for web applications implemented in the Spring framework in Java, using aspect oriented programming and a pair of alternatives. The first solution involves a few modifications to the engine's source code, whereas the second does not. Table 6.2 summarizes the changes made to those software packages.

The implementation of query identifiers has to be compatible with all the components we have been discussing: application source code, SSLE, and DBMS. Specifically, it is important that having SEPTIC in the DBMS or generating IDs in the SSLE does not require modifications to the other components. The solution is to place the identifiers inside DMBS comments. SEPTIC assumes that the first comment in a query is the ID. We place the comment at the beginning of the query, before the query proper.

6.3.1 Protecting MySQL

We implemented SEPTIC – i.e., the center and right-hand side of Figure 6.1(b) – in MySQL 5.7.4. We modified a single file of the package (`sql_parser.cc`) and added a new header file (SEPTIC detector) and a configuration file (SEPTIC setup), plus the plugins, which are external to the DBMS (e.g., for stored XSS the plugin is essentially the *jsoup* library (JSoup,

Software	sfm	sfc	loc	sa
MySQL 5.7.4				
- `sql_parser.cc`	1	–	14	–
- SEPTIC detector	–	1	1570	plugins
- SEPTIC setup	–	1	15	–
- *septic_training*	–	1	380	–
Zend engine / PHP 5.5.9				
- mysql extension	1	–	6	–
- mysqli extension	2	–	21	–
- SEPTIC identifier	–	1	249	–
Spring 4.0.5 / Java				
- `JdbcTemplate.java`	1	–	16	–
- SEPTIC identifier	–	1	–	–

sfm: source file modified — loc: lines of code
sfc: source file created — sa: software added

Table 6.2: Summary of modifications to software packages

2014)). The *septic_training* module is not only external but also runs separately from the DBMS.

The lines added to the `sql_parser.cc` file were inserted in function `mysql_parse`, and just before the call to the function `mysql_execute_command` that executes the query. These lines call the SEPTIC detector with an input corresponding to the query parsed and validated by MySQL. The module performs the previously described operations: builds the query structure (QS); compares QS with its query model (QM); logs the query and the ID if an attack is detected; and optionally drops the query.

In detail, the SEPTIC detector is executed by the `compareQueryStructure` function. This function calls the `processSelect_Lex` and `insertElementTemplate` functions to check the query command (`SELECT, DELETE, INSERT, UPDATE`) and to build the QS. At the same time, this function gets the query ID and verifies if a QM for that query ID exists in *query models* storage. If QM exists, it is loaded and the `compareQueryToTemplate` function is called to compare QS with QM. Otherwise, the QM is built from the QS, and then it is associated to that query ID and it is stored in the *query models* storage (training *incremental* method). Comparing QS with QM means to perform the algorithm of detection for SQLI attacks presented in Section 6.2.4. First, it verifies if the number of items in both stacks is equal (structural verification). If it is not, a SQLI attack is detected. Otherwise, the syntactical verification is performed. For this verification, the `processItem` function is called to compare each item of the QS with its correspondent item of the QM. If any

comparison does not match, a SQLI attack is flagged. The `processItem` function analyzes 27 different types of items defined in the MySQL. The function uses two auxiliary functions – `processField` and `isPrimitiveTypeBenign` – to detect differences between fields and to check if an item is a primitive data type (integer, real, string or decimal), allowing casts between them.

The detection of stored injection attacks is made for `INSERT`, and `UPDATE` SQL commands and performed by the `processItem` function. The function performs the filtering and testing steps explained in Section 6.2.4 for the string items contained in the queries.

SEPTIC is configured using a few switches. The first allows putting SEPTIC in training mode, detection mode (logs attacks), or prevention mode (logs and blocks attacks). The other two enable and disable respectively the detection of SQLI and stored injection attacks. The values for these switches are defined in a configuration file (SEPTIC setup) that is read by MySQL whenever it is started or restarted. A typical routine consists in setting the first switch to training mode and the other two switches to *on*. Then, the DBMS and the web server are initiated, running the *septic_training* module. Later, the first switch is changed to prevention or detection mode, followed by the restart of the DBMS and the application server.

6.3.2 Inserting identifiers in Zend

In Section 6.2.3 we discussed three kinds of IDs. We implemented the first kind – *SSLE-generated IDs* – for the PHP language, with the Zend engine as SSLE. As explained in that section, those IDs can be formed by pairs of *file:line* separated by |. So the comments we consider in this section have the format `/* file:line | file:line | ... | file:line */`.

Table 6.2 shows the two Zend engine extensions to which we added a few lines of code to create and insert query IDs. Extensions are used in Zend to group related functions. The table shows also the new header file that we developed for the same purpose (SEPTIC identifier).

The identifiers have to be inserted when the DBMS is called, so we modified in Zend the 11 functions used for this purpose (e.g., `mysql_query`, `mysqli::real_query`, and `mysqli::prepare`). Specifically, the ID is inserted in these functions just before the line

that passes the query to the DBMS. This involved modifying three files: `php_mysql.c`, `mysqli_api.c` and `mysqli_nonapi.c`.

In detail, the `generate_ID` function is implemented in Zend by our `get_query_ID` function that calls other three functions: `get_function_args`, `get_query_index` and `get_query_function_args`. Listing 6.2 presents the algorithm to get the query ID implemented by the `get_query_ID` function.

```
1  ID
2  backtrace = true
3  while backtrace and not empty stack do
4      TOP function from stack
5      get function_name
6      get array_args_function
7      get filename_function_call
8      get line_function_call
9      compose pair filename_function_call:line_function_call
10     concatenate pair to previous ID
11     if function_name equals some sensitive sink then
12         get query
13     else
14         if query not in array_args_function then
15             backtrace = false
16         end_if
17     end_if
18     compose pair filename_function_call:line_function_call
19     concatenate pair to previous ID
20     POP function from the stack
21 end_do
```

Listing 6.2: Algorithm to get the query ID.

When a PHP program is executed, Zend keeps a call stack. This stack contains data about the functions called, such as function name, full pathname of the file and line of code where the function was called. This stack allows backtracking the query until a function that does not contain it as argument. This provides the places where the query was composed and/or was argument of a function, and allows obtaining query IDs in the format above.

The algorithm presented in Listing 6.2 represents this backtrack and composition of the backtrace of the query. A TOP stack operation is made, accessing thus the call function in the top of the stack. The information listed above is retrieved (lines 4 to 8) to compose the pair

file:line and concatenate it with the previous ID, resulting a new ID (lines 9 and 10). Then, if the function is a sensitive sink we get the query argument to start backtracking it (lines 11 and 12). Otherwise, the algorithm checks if the query belongs to the array of the function arguments. If not, the backtracking stops, otherwise a POP stack operation is made (line 18) and a new loop iteration is performed.

6.3.3 Inserting identifiers in Spring / Java

We also implemented the third kind of IDs explained in Section 6.2.3 – *IDs generated outside the DBMS and the SSLE* – in Spring / Java. Spring is a framework aimed at simplifying the implementation of enterprise applications in the Java programming language (Spring, 2014a). It allows building Java web applications using the Model-View-Controller (MVC) model. In Spring applications connect to the DBMS via a JDBC driver.

We used three different forms to insert the IDs to show the flexibility of doing it. The first form consists in inserting the ID directly in the query in the source code of the application. Before the query is issued a comment with the ID is inserted. This is a very simple solution that has the inconvenient of requiring modifications to the source code. The second form uses a *wrapper* to catch the query request before it is sent to JDBC and MySQL, and insert the ID in a comment prefixing the query (e.g., the file and line data). Using a wrapper avoids the need to modify the source code of the application, except for the substitution of the calls to JDBC by calls to the wrapper.

The third form is the most interesting as it does not involve modifications to the application source code. We use *Spring AOP*, an implementation of Aspect-Oriented Programming for Spring, essentially to create a wrapper without modifying the applications' source code (Spring, 2014b). Spring AOP allows the programmer to create *aspects* for the application that he is developing. These aspects allow the interception of method calls from the application, to insert code that is executed before the methods. These operations are performed without the programmer making changes to the application source code. On the contrary, the programmer develops new files with the *aspects* and their *point cuts*, where the point cuts are the application methods that will be intercepted. We use aspects for intercepting in runtime calls to JDBC, inserting the query ID in the query and proceeding with the query request to MySQL.

6.4 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions:

1. Is SEPTIC able to detect and block attacks against code samples?
2. Is it more efficient than other tools in the literature?
3. Does it solve the semantic mismatch problem better than other tools?
4. How does it perform in terms of false positives and false negatives?
5. Is SEPTIC capable of discovering and blocking attacks against real (open source) software?
6. Is the performance overhead acceptable?

The evaluation was carried out with the implementation of SEPTIC in MySQL and PHP/Zend. Sections 6.4.1 presents the evaluation of SEPTIC in terms of its ability to detect attacks – questions 1 to 5 – and Section 6.4.2 presents the evaluation of performance overhead – question 6.

6.4.1 Attack detection

Detection with code samples

To answer questions 1. to 4., we evaluated SEPTIC with:

1. a set of (small) code samples that perform attacks of all classes in Table 6.1 (17 for the semantic mismatch problem, 7 for other SQLI attacks, 5 for stored injection);
2. 23 code samples from the *sqlmap* project (sqlmap, 2014), unrelated with semantic mismatch;
3. 11 samples with the code and non-code injection cases defined in (Ray & Ligatti, 2012) (Table 6.3).

	Case	Attack/code
1	`SELECT balance FROM acct WHERE password='' OR 1=1 -- '`	Yes
2	`SELECT balance FROM acct WHERE pin= exit()`	Yes
3	`...WHERE flag=1000>GLOBAL`	Yes
4	`SELECT * FROM properties WHERE filename='f.e'`	No
5	`...pin=exit()`	Yes
6	`...pin=aaaa()`	Yes
7	`SELECT * FROM t WHERE flag=TRUE`	No
8	`...pin=aaaa()`	Yes
9	`SELECT * FROM t WHERE password=password`	Yes
10	`CREATE TABLE t (name CHAR(40))`	No
11	`SELECT * FROM t WHERE name='x'`	No

Table 6.3: Code (attacks) and non-code (non-attacks) cases defined by Ray and Ligatti (Ray & Ligatti, 2012). Although those authors consider case 10 to be code/attack, we disagree because the input is an integer, which is the type expected by the *char* function.

We compare SEPTIC with a Web Application Firewall (WAF) and four anti-SQLI tools. Figure 6.6 shows the place where the WAF and the anti-SQLI tools act and intercept, respectively, the user inputs sent in HTTP requests and the query sent by the web application. SEPTIC acts inside the DBMS. The WAF was ModSecurity 2.7.3.3 (Trustwave SpiderLabs, 2015), which was configured with the OWASP Core Rule Set 2.2.9. ModSecurity is the most adopted WAF worldwide, with a stable rule set developed by experienced security administrators. In fact, it has been argued that its ability to detect attacks is hard to exceed (Modelo-Howard *et al.*, 2014). It detects SQLI and other types of attacks by inspecting HTTP requests. The anti-SQLI tools were: CANDID (Bandhakavi *et al.*, 2007), AMNESIA (Halfond & Orso, 2005), DIGLOSSIA (Son *et al.*, 2013) and SQLrand (Boyd & Keromytis, 2004). The evaluation of these tools was made manually by analyzing the data in (Ray & Ligatti, 2012) and the papers that describe them. More information about them can be found in Section 2.4.2.

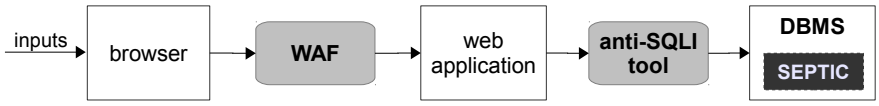


Figure 6.6: Placement of the protections considered in the experimental evaluation: SEPTIC, anti-SQLI tools, and a WAF.

In the experiments, first with SEPTIC turned *off*, we injected malicious user inputs created manually to confirm the presence of the vulnerabilities in the code samples. We also used the *sqlmap* tool to exploit the vulnerabilities from the first two groups of code samples. *sqlmap* is a tool widely used to perform SQLI attacks, both by security professionals and hackers. Second, with SEPTIC turned *on* and in training mode, we injected benign inputs in the code samples for the mechanism to learn the queries and to get their models. Then, we run the same attacks from the first phase in detection mode and analyzed the results to determine if they were detected.

Table 6.4 shows the results of the evaluation. There were 63 tests executed (third column), 4 of which not attacks (the 4 non-attack cases in Table 6.3). SEPTIC (last column) correctly detected all 59 attacks (row 34) and correctly did not flag as attacks the 4 non-attack cases (row 11). SEPTIC had neither false negatives nor positives (rows 35–36) and correctly handled the semantic mismatch problem by detecting all attacks from classes A (rows 17–21), B (7), C (8–9), and D.2–D.4 (26–30).

The other tools can also detect the syntax structure 1st order (row 3), blind SQLI syntax structure (8), and sqlmap (12) attacks (all from class S.1), but not stored procedure (7) and stored injection attacks (26–30). The anti-SQLI tools, only found the attack from class A.5 in the semantic mismatch attacks (row 21). ModSecurity detected this attack plus 1st order SQLI attacks with encoding and space evasion (A.1 and A.4, rows 17 and 19). Furthermore, ModSecurity could not detect 2nd order SQLI attacks because in the second step of these attacks the malicious input comes from the DBMS, not from outside. All tools other than SEPTIC had a few false positives (except DIGLOSSIA) and many false negatives (around 50% of the attacks). This is essentially justified by the non-detection of semantic mismatch attacks and the Ray and Ligatti code cases (row 10), where the injected code does not contain malicious characters recognized by the tools.

Globally ModSecurity and DIGLOSSIA had a similar performance (35 attacks detected). The latter was the best of the four anti-SQLI tools and the only one that detected the syntax mimicry 1st order attack (row 5). ModSecurity does not flag 2nd order attacks because it just analyses queries with user inputs (rows 18 and 20). On the contrary, SQLrand and AMNESIA detect this type of attack. CANDID does not discover either of them. The false positive reported for ModSecurity was case 11 from (Ray & Ligatti, 2012), as the input contained the prime character that is considered malicious by this WAF.

1	Type of attack	N. Tests	SQLrand	AMNESIA	CANDID	DIGLOSSIA	ModSecurity	SEPTIC
2	**SQLI without sanitization and semantic mismatch (S.1, S.2, B, C, D.1)**							
3	Syntax structure 1st order	1	Yes	Yes	Yes	Yes	Yes	Yes
4	Syntax structure 2nd order	1	Yes	Yes	No	No	No	Yes
5	Syntax mimicry 1st order	1	No	No	No	Yes	Yes	Yes
6	Syntax mimicry 2nd order	1	No	No	No	No	No	Yes
7	Stored procedure	1	No	No	No	No	No	Yes
8	Blind SQLI syntax structure	1	Yes	Yes	Yes	Yes	Yes	Yes
9	Blind SQLI syntax mimicry	1	No	No	No	Yes	Yes	Yes
10	Ray & Ligatti code	7	2	3	3	7	2	7
11	Ray & Ligatti non-code	4 (non-attacks)	2	1	2	0	1	0
12	sqlmap project	23	23	23	23	23	23	23
13	Flagged as attack	–	30	30	30	34	30	37
14	False positives	–	2	1	2	0	1	0
15	False negatives	–	9	8	9	3	8	0
16	**SQLI with sanitization and semantic mismatch (S.1, S.2, A.1–A.5, D.1)**							
17	Syntax structure 1st order	4	0	0	0	0	2	4
18	Syntax structure 2nd order	4	0	0	0	0	0	4
19	Syntax mimicry 1st order	4	0	0	0	0	2	4
20	Syntax mimicry 2nd order	4	0	0	0	0	0	4
21	Numeric fields	1	1	1	1	1	1	1
22	Flagged as attack	–	1	1	1	1	5	17
23	False positives	–	0	0	0	0	0	0
24	False negatives	–	16	16	16	16	12	0
25	**Stored injection (D.2–D.4)**							
26	Stored XSS	1	No	No	No	No	No	Yes
27	RFI	1	No	No	No	No	No	Yes
28	LFI	1	No	No	No	No	No	Yes
29	RCI	1	No	No	No	No	No	Yes
30	OSCI	1	No	No	No	No	No	Yes
31	Flagged as attack	–	0	0	0	0	0	5
32	False positives	–	0	0	0	0	0	0
33	False negatives	–	5	5	5	5	5	0
34	**Flagged as attack**	–	31	31	31	35	35	59
35	**False positives**	–	2	1	2	0	1	0
36	**False negatives**	–	30	29	30	24	25	0

Table 6.4: Detection of attacks with code samples.

The answer to the first four questions is positive. We conclude that the proposed approach to detected and block injection attacks inside the DBMS is effective because it uses the information given by the DBMS – that processes the queries – without the need of assumptions about how the queries are executed, which is the root of the semantic mismatch problem.

Detection with real software

We used SEPTIC with real web applications to verify if it identifies attacks against them – question 5. We evaluated it with five open source PHP web applications: *ZeroCMS*, a content management system (ZeroCMS, 2014); *WebChess*, an application to play chess online

(WebChess, 2014); *measureit*, an energy metering application that stores and visualizes voltage and temperature data (Measureit, 2014); *PHP Address Book*, a web-based address/phone book (PHP Address Book, 2015); and *refbase*, a web reference database (refbase, 2015).

Table 6.5 shows the detection results. The *ZeroCMS* contains three SQLI vulnerabilities that appeared in the Common Vulnerabilities and Exposures (CVE) (CVE, 2015) and the Open Source Vulnerability Database (OSVDB) (OSVDB, 2015): CVE-2014-4194, CVE-2014-4034 and OSVDB ID 108025. Using *sqlmap*, we performed SQLI attacks to exploit these vulnerabilities and to verify if SEPTIC detected them. SEPTIC successfully found the attacks and blocked them, protecting the vulnerable web application. Also, we performed attacks against a patched version of *ZeroCMS* and verified that the attacks were no longer successful or detected by SEPTIC.

With *WebChess* and *measureit*, we performed attacks manually and with *sqlmap*. SEPTIC blocked 13 different attacks against *WebChess* and one stored XSS against *measureit*. To confirm the detection, we repeated the attacks with SEPTIC in detection mode (instead of prevention mode), allowing attack discovering but without blocking them, and we verified their impact. Also, we confirmed the vulnerabilities explored by these attacks by inspecting the source code with the assistance of identifiers registered in the log file. We recall that our approach identifies in runtime attacks and registers the source code location of the vulnerabilities explored by attacks when they are detected. SEPTIC does not registered any attack against the *PHP Address Book* and *refbase* applications, meaning that these applications are secure against attack injection. So these results allow us to answer affirmatively to question 5.

Application	**SQLI**	**Stored inj.**	**Registered**
measureit	–	1	–
PHP Address Book	–	–	–
refbase	–	–	–
WebChess	13	–	–
ZeroCMS	3	–	CVE-2014-4194 CVE-2014-4034 OSVDB ID 108025
Total	16	1	3

Table 6.5: Detection of attacks in real applications

6.4.2 Performance overhead

To answer question 6., we evaluated the overhead of SEPTIC using BenchLab v2.2 (Cecchet *et al.*, 2011) with the *PHP Address Book*, *refbase* and *ZeroCMS* applications. BenchLab is a testbed for web application benchmarking. It generates realistic workloads, then replays their traces using real web browsers, while measuring the application performance.

We have set up a network composed of six identical machines: Intel Pentium 4 CPU 2.8 GHz (1-core and 1-thread) with 2 GB of RAM, running Linux Ubuntu 14.04. Two machines played the role of servers: one run the MySQL DBMS with SEPTIC; the other executed an Apache web server with Zend and the web applications, and Apache Tomcat to run the BenchLab server. The other four machines were used as client machines, running BenchLab clients and Firefox web browsers to replay workloads previously stored by the BenchLab server, i.e., to issue a sequence of requests to the web application being benchmarked. The BenchLab server has te role of managing the experiments.

We evaluated SEPTIC with its four combinations of protections turned on and off (SQLI and stored injection on/off) and compared them with the original MySQL without SEPTIC installed (base). For that purpose, we created several scenarios, varying the number of client machines and browsers. The *ZeroCMS* trace was composed of 26 requests to the web application with queries of several types (`SELECT`, `UPDATE`, `INSERT` and `DELETE`). The traces for the other applications were similar but for *PHP Address Book* the trace had 12 requests, while for *refbase* it had 14 requests. All traces involved downloading images, cascading style sheets documents, and other web objects. Each browser executes the traces in a loop many times.

Table 6.6 summarizes the performance measurements. The main metric assessed was the *latency*, i.e., the time elapsed between the browser starts sending a request and finishes receiving the corresponding reply. For each configuration the table shows the *average latency* and the *average latency overhead* (i.e., the average latency divided by the latency obtained with MySQL without SEPTIC with the same configuration, multiplied by 100 to become percentage). These values are presented as a pair *(latency (ms), overhead (%))* and are shown in the 2nd to 6th columns of the table. The first column characterizes the scenario tested, varying the number of client machines (*PCs*) and browsers (*brws*). The latency obtained with MySQL without SEPTIC is shown in the second column and the SEPTIC combinations in the next four. The last two columns show, respectively, the number of times that each

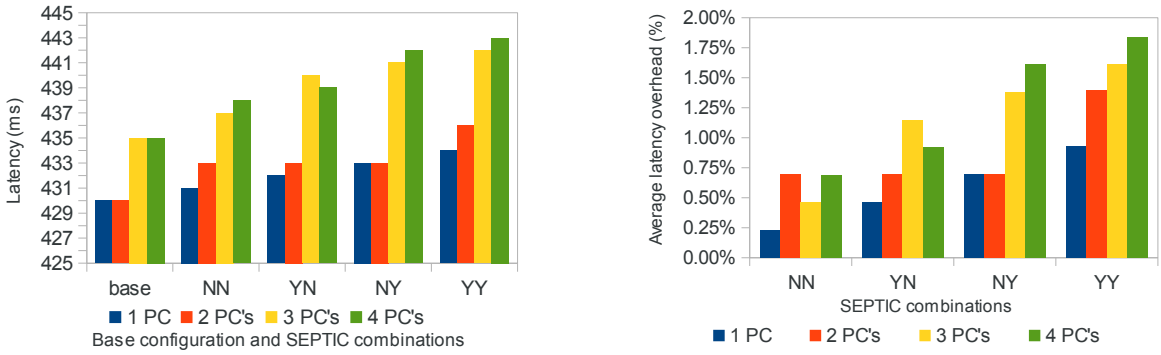
configuration was tested with a trace (*num exps*) and the total number of requests done in these executions (*total reqs*). Each configuration was tested with 5500 trace executions, in a total of 87,200 requests (last row of the table).

N. PCs & brws	Base	SEPTIC: SQL injection – stored injection				Num exps	Total reqs
		off–off	on–off	off–on	on–on		
***refbase* varying the number of PCs, one browser per PC**							
1 PC	430, –	431, 0.23	432, 0.47	433, 0.70	434, 0.93	70	980
2 PCs	430, –	433, 0.70	433, 0.70	433, 0.70	436, 1.40	120	1680
3 PCs	435, –	437, 0.46	440, 1.15	441, 1.38	442, 1.61	170	2380
4 PCs	435, –	438, 0.69	439, 0.92	442, 1.61	443, 1.84	220	3080
***refbase* with four PCs and varying the number of browsers**							
8 brws	504, –	506, 0.40	510, 1.19	513, 1.79	516, 2.38	420	5880
12 brws	530, –	532, 0.38	535, 0.94	539, 1.70	544, 2.64	620	8680
16 brws	540, –	541, 0.19	545, 0.93	550, 1.85	553, 2.41	820	11480
20 brws	570, –	573, 0.53	575, 0.88	581, 1.93	584, 2.46	1020	14280
***PHP Address Book* with four PCs**							
20 brws	79, –	79.26, 0.33	79.50, 0.63	80.60, 2.03	81, 2.53	1020	12240
***ZeroCMS* with four PCs**							
20 brws	239, –	240, 0.42	241, 0.84	243, 1.67	245, 2.51	1020	26520
Avg. overhead / Total		*0.41%*	*0.82%*	*1.65%*	*2.24%*	5500	87200

Table 6.6: Performance overhead of SEPTIC measured with Benchlab for three web applications: *PHP Address Book*, *refbase* and *ZeroCMS*. Latencies are in milliseconds and overheads in percentage.

The first set of experiments evaluated the overhead of SEPTIC with the *refbase* application (rows 3–6). We run a single Firefox browser in each client machine but varied the number of these machines from 1 to 4. For each additional machine we increase the number of experiments (*num exps*) by 50. Figure 6.7 represents graphically these results, showing the latency measurements (a) and the latency overhead of the different SEPTIC configurations (b). The most interesting conclusion taken from the figure is that the overhead of running SEPTIC is very low, always below 2%. Another interesting conclusion is that SQLI detection has less overhead than stored injection detection, as the values for configuration NY are just slightly higher than those for YN. Finally, it can be observed that the overhead tends to increase with the number of PCs and browsers generating traffic as the load increases.

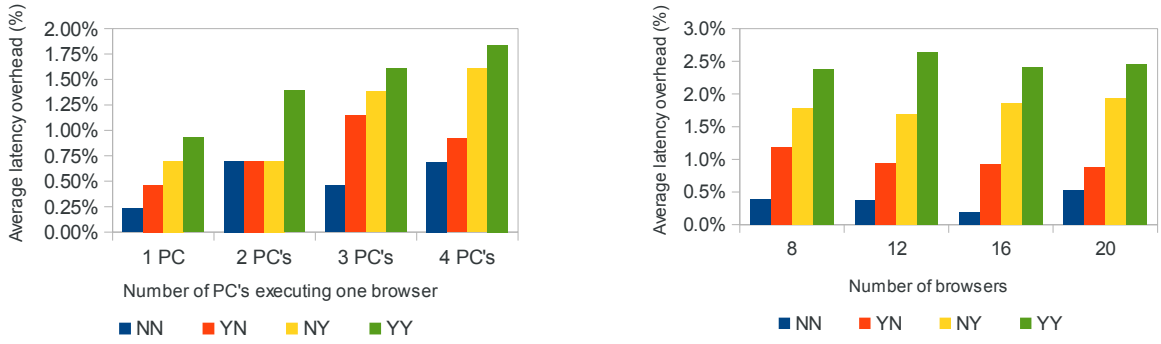
The second set of experiments used again the *refbase* application, this time with the number of client machines (PCs) set to 4 and varying the number of browsers (Table 6.6, rows 8–11). Figure 6.8 shows how the overhead varies when going from 1 to 4 PCs with one



(a) Latency (b) Overhead

Figure 6.7: Latency and overhead with *refbase* varying the number of PCs, each one with a single browser.

browser each (a) then from 8 browsers (2 per PC) to 20 browsers (5 per PC). The figure allows extracting some of the same conclusions as the first set of experiments. However, they also show that increasing the number of browsers initially increases the overhead (Figure 6.8(a)), then stabilizes (b), as neither the CPU at the PCs nor the bandwidth of the network were the performance bottleneck.



(a) Overhead (one browser per PC) (b) Overhead (varying browsers)

Figure 6.8: Overhead with *refbase* with 4 PCs and varying the number browsers.

The third and fourth sets of experiments used the *PHP Address Book* and *ZeroCMS* web applications and 20 browsers in 4 PCs (Table 6.6, rows 13 and 15). Figure 6.9 shows the overhead of these two applications and *refbase* with the same number of browsers and PCs. The overhead of all applications is similar for each SEPTIC configuration. This is interesting because the applications and their traces have quite different characteristics, which suggests

that the overhead imposed by SEPTIC is independent of the server-side language and web application.

The average of the overheads varied between 0.82% and 2.24% (last row of the table). This seems to be a reasonable overhead, suggesting that SEPTIC is usable in real settings, answering positively question 6..

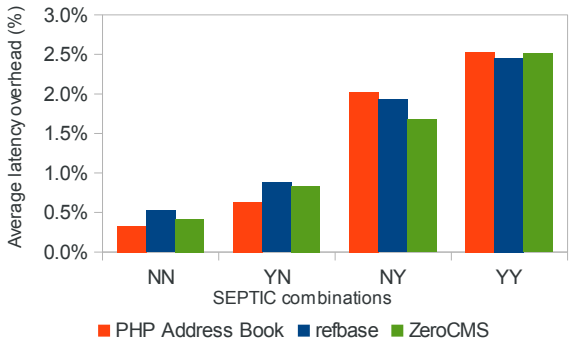


Figure 6.9: Overhead of SEPTIC with *PHP Address Book*, *refbase* and *ZeroCMS* applications using 20 browsers.

6.5 Extensions to SEPTIC

The previous sections presented the SEPTIC approach, its implementation, and experimental evaluation. This section discusses extensions to the core approach and implementation.

6.5.1 Protecting other DBMSs

The SEPTIC approach is supposed not to be restricted to work with MySQL. To show that this is the case, we discuss how to implement the approach in two other DBMSs, based on an analysis we have made of their source code. We analyzed MariaDB 10.0.20 (MariaDB, 2015) and PostgreSQL 9.4.4 (PostgreSQL, 2015). MariaDB is a fork of MySQL created around 2009 due to concerns over Oracle's acquisition of MySQL. PostgreSQL is the second most popular open source DBMS, after MySQL (DB-Engines, 2015).

6.5.1.1 MariaDB

MariaDB has essentially the same architecture as MySQL. When a query is received, it parses, validates, and executes it (see Figure 6.1(a)). The outcome of the parsing and validation phases is the same as in MySQL, a *list of stacks* where each stack of the list represents a clause of the query, and each of its nodes contains data about the query element. Moreover, the file that contains the calls to the functions that perform parsing, validation and execution of a query is the same as in MySQL: `sql_parser.cc`. Therefore, SEPTIC can be implemented in MariaDB similarly to how it was in MySQL (Section 6.3.1). The SSLE-generated IDs implemented in the Zend engine can be used without any modification.

6.5.1.2 PostgreSQL

The implementation of SEPTIC in PostgreSQL has some differences but also many similarities to the MySQL and MariaDB cases. The processing of a query in PostgreSQL involves four phases: parsing/validation, rewriting, planning/optimization, and execution. Again the SEPTIC module is inserted after the parsing phase, before the rewriting phase. Similarly to MySQL, a single file has to be modified (`postgresql.c`), adding essentially the same 14 lines of code that were added to MySQL. That file contains the function `exec_simple_query` that runs the four processing phases of a query. The code would be inserted after the call to function `pg_parse_query` that parses and validates the query, just before the call to the function that executes the rewriting phase (`pg_analyze_and_rewrite`). SEPTIC might also be inserted after the rewriting phase, but the adaptation would be more complicated as rewriting produces a different data structure, a query tree.

The data structure resulting from the parsing phase is slightly different from MySQL's but still a *list of stacks*. Again each stack of the list represents a clause of the query (e.g., `SELECT`, `FROM`) and its nodes a query element. PostgreSQL tags the query elements with their types and distinguishes the primitive types (e.g., integer, float/real, string). The nodes of the stacks contain this information similarly to what happens in MySQL, but the tags, the structure of the nodes, and the way they are organized in the stack are different from MySQL. Therefore, the data structures used in PostgreSQL and MySQL are similar, but the current implementation of the module *SEPTIC detector* has to be modified, specifically: (1) the navigation in the *list of stacks*; (2) the identification of the data about the query elements

in the nodes; and (3) the collection of this data. These modifications are related with the construction of query structure for every query.

Similarly to MariaDB, no changes are needed to the generation of IDs implemented in the Zend engine.

6.5.2 Vulnerability diagnosis

SEPTIC aims to protect web applications transparently from the administrator or programmer. However, if an attack is successful that means there is a vulnerability in the application and it may be useful to understand where that vulnerability is in the source code.

SEPTIC combined with the SSLE-generated ID presented in Section 6.2.3 can provide this information when an attack is detected. Recall that the SSLE-generated ID we propose contains information about the places in the source code where the query is passed to functions as a parameter and eventually sent to the DBMS. These places are identified by the *file:line* pairs in the ID, as explained in Section 6.2.3. When SEPTIC detects an attack it logs the ID and the query, both in detection and prevention modes.

The administrator/programmer can use this log to diagnose the vulnerability. Moreover, it can use the attack query to understand how the vulnerability was exploited and how it can be removed. Some rules of thumb on how to fix the application are:

- SQLI attack and user inputs are not sanitized: any of the attacks of Table 6.1 may have happened; sanitization and/or validation has to be inserted in the source code;
- SQLI attack and user inputs are sanitized: the attack probably belonged to class A, possibly a case of semantic mismatch; proper sanitization or validation has to be implemented to deal with these attacks;
- Second order SQLI attack: the query contains inputs provided by another query, thus introduced earlier in the database; therefore, the inputs provided by the other query have to be sanitized/validated;
- Stored injection: the attack belonged to classes D.2–D.4; the programmer has to develop validation routines to apply to the inputs.

6.5.3 Detecting attacks against non-web applications

Despite the chapter has focused on the detection of injection attacks against web applications, the SEPTIC approach (and its implementation in MySQL) also works with non-web applications. DBMSs are mostly oblivious to the type of applications that send them requests. The SEPTIC module inside the DBMS is also oblivious to the applications, except for query IDs. However, queries do not have to bring an ID as SEPTIC can also use *DBMS-generated IDs* (see Section 6.2.3).

Attacks coming from non-web applications can be detected by SEPTIC using such *DBMS-generated IDs* or *IDs generated outside the DBMS and the SSLE*. Similarly to what happens with web applications, SEPTIC has to undergo training to learn about the normal queries issued by the (non-web) application, in order to build their query models. This training cannot be done with the *septic_training* module, which is specific for web applications, but the idea is the same: to activate all queries with good inputs.

With the goal of demonstrating that SEPTIC also works with non-web applications, we developed a simple Gambas application to manage contacts, i.e., an address book (Gambas, 2015). Gambas is a version of .NET / Visual Basic for Linux. Applications in Gambas can connect to a DBMS and issue requests similarly to what happens in PHP and Java. We trained SEPTIC using the incremental method (Section 6.2.5), i.e., by forcing the application to issue non-malicious queries to the database without putting SEPTIC in training mode. Then, we injected a few attacks that SEPTIC correctly detected.

6.6 Conclusions

The chapter explores a new form of protection from attacks against web applications that use databases. It presents the idea of "hacking" the DBMSs to let it protected from SQLI and stored injection attacks. Moreover, by putting protection inside the DBMS, we show that it is possible to detect and block sophisticated attacks, including those related with the semantic mismatch problem.

All the works we describe in Section 2.4.2 have a point in common that makes them quite different from our work: their focus is on *how to do detection or protection*. On the contrary, our work is more concerned with an architectural problem: *how to do detection/protection inside the DBMS*, so that it runs out of the box when the DBMS is started. None of the related

works does detection inside the DBMS. Also, those schemes cannot deal with most attacks related with the semantic mismatch problem. SEPTIC, on the contrary, does not involve source code analysis or instrumentation, or modifying the application code. With SEPTIC we aim to make the DBMS protect itself, so that both the model creation and attack detection are performed inside the DBMS. Moreover, SEPTIC aims to handle the semantic mismatch problem, so it analyses queries just before they are executed, whereas these works do it much earlier. Most of these works also cannot detect attacks that do not change the structure of the query (syntax mimicry), unlike SEPTIC. For example, AMNESIA and CANDID are two of them. SqlCheck detects some of the attacks related with semantic mismatch, but not those involving encoding and evasion. Like SEPTIC, DIGLOSSIA detects syntax structure and mimicry attacks but, unlike SEPTIC, it neither detects second-order SQLI once it only computes queries with user inputs, nor encoding and evasion space characters attacks as these attacks do not alter the parse tree root nodes before the malicious user inputs are processed by the DBMS.

The SEPTIC mechanism was experimented both with synthetic code with vulnerabilities inserted on purpose, including a set of novel SQLI attacks presented recently (Ray & Ligatti, 2012), and with open source PHP web applications. This evaluation suggests that the mechanism can detect and block the attacks it is programmed to handle, performing better that all other solutions in the literature, anti-SQLI mechanisms and the ModSecurity WAF. SEPTIC shows neither false negatives nor false positives, on the contrary of the others. The performance overhead evaluation shows an impact of around 2.2%, suggesting that our approach can be used in real systems.

7

Conclusions and Future Work

This thesis proposes methodologies to detect and locate input validation vulnerabilities in source code of web applications, exploring source code static analysis, machine learning and runtime protection techniques. Static analysis and runtime protection are used as two distinct ways to address vulnerabilities. While the former analyzes the source code of web applications to search for vulnerabilities, the latter monitors in runtime the web applications to block injection attacks, which in conjunction with identifiers allows the detection of vulnerabilities. Machine learning is applied with and as a static analysis technique to reduce the number of false positives and to find vulnerabilities. Furthermore, the methodologies described provide protection for web applications, removing the vulnerabilities by automatic correction of source code, and blocking injection attacks that attempt to exploit vulnerabilities contained in the source code.

The chapter is divided in two sections that present the conclusions and future work that could derive from this thesis.

7.1 Conclusions

The thesis begins by showing that input validation vulnerabilities are an important problem in web applications, and how they can be detected and removed in source code. Then, it presents the methodologies to detect and eliminate those vulnerabilities and to protect web applications.

Taint analysis is a form of static analysis that can be used to verify the source code of web applications looking for input validation vulnerabilities, i.e., tracking the user inputs (entry points) and checking if they reach a function susceptible to be exploited (a sensitive sink). The thesis shows that this technique is effective to search for vulnerabilities in source code. However, static analysis tends to generate many false positives, so we propose the use of machine learning applied to data mining to reduce this tendency. Thus, the vulnerabilities found by taint analysis are processed by data mining, predicting if they are false positives or not. This form of analysis turns the process of detection more accurate. Benefiting from the identification, i.e., of the localization of the vulnerabilities in the source code, a step further is made by the automatic correction of the source code. Using the identification provided by the taint analysis, we correct the source code by inserting fixes in the right places of the program to sanitize and validate the entry points. This is an important contribution because it helps the programmers in different ways by: (1) verifying the code while the applications are being developed; (2) signaling in the source code the location of the vulnerabilities, to remove them automatically by correcting the source code; (3) avoiding the waste of time checking the source code for vulnerabilities that are not real, i.e., looking for false positives; (4) keeping the programmers in the loop of vulnerability detection and correction, showing them the vulnerable code and how it is corrected.

Another contribution of the thesis is a novel source code static analysis technique to find vulnerabilities and their location in source code. Apart from the traditional static analysis technique and the standard machine learning classifiers used in data mining, we propose an approach that uses sequence models with machine learning to obtain knowledge about vulnerabilities, learning their characteristics from a corpus with sequences of observations annotated as vulnerable or not. Then, new sequences of observations are processed and classified as vulnerable or not. The sequence model is a hidden Markov model that processes the source code (sequences of observations) taking in consideration the order of code elements (observations) inside the code and the relation between them.

This novel technique improves the first one in two ways. One is the ability to relate the code elements, which allows a more sophisticated analysis. Previously, it only checked the presence of characteristics about false positives and vulnerabilities in source code and did not relate them. The other is the absence of coding these characteristics and their relationship, i.e, there is no need to coding the knowledge about vulnerabilities.

Another important contribution of the thesis is a runtime protection for web applications to block injection attacks. This approach helps to solve the semantic mismatch problem, and interestingly also supports the discovery of vulnerabilities, identifying their location in source code. Contrarily to the other two approaches that analyze the source code, we propose to monitor the web applications in runtime and use identifiers that carry information about the location of queries composition through the code until the sensitive sinks. Therefore, when an injection attack is identified this information is used to identify the vulnerability. Another contribution is the place where the mechanism that implements the approach is inserted. We opted by the DBMS, "hacking" it. The DBMS is the entity that does the final processing of the queries, so inserting there the protection solves the semantic problem, since at that point there is not speculation about how the queries end up being executed.

The methodologies mentioned above resulted in the development of two tools and a mechanism. The WAP and DEKANT tools implement the first two methodologies, while the SEPTIC mechanism implements the last.

The WAP tool detects candidate vulnerabilities using taint analysis, predicts if they are real vulnerabilities or false positives using data mining, and corrects the source code by insertion of fixes. It was experimented in two different stages of its development. In its first stage, the tool was able to detect eight classes of vulnerabilities, and our evaluation with open source web applications confirmed that WAP detects and corrects vulnerabilities and reduces the number of false positives, when compared with other tools. Recently, we evolved the tool changing its structure to make it modular and extensible for new vulnerability classes, without the programmer coding about the new classes. The three components of the tool that perform the detection, prediction and correction were re-structured to enable the automatic creation and setting up of WAP extensions (called *weapons*). The data mining component was also enhanced with new attributes and instances. The tool, in this stage, was evaluated with fifteen classes of vulnerabilities (seven are weapons) using real web applications and WordPress plugins. The results confirmed the benefits of re-structuration compared with the tool in its first stage.

DEKANT is a tool that learns about vulnerabilities and then detects them. The tool extracts slices, starting in entry points and ending in sensitive sinks, translates them to an intermediate language, and next classifies the new slices as being vulnerable or not using a sequence model implemented by an HMM. It was experimented with open source software

and WordPress plugins. Both experiments confirmed the effectiveness of the tool, meaning that it is feasible to learn about vulnerabilities and afterwards apply this knowledge to search for vulnerabilities.

Finally, the SEPTIC mechanism, inserted inside the DBMS, has the aim to protect the web applications that it monitors. SEPTIC requires a training phase to learn the query models from the web application to be monitored, and then during the detection phase flags injection attacks and identifies the vulnerabilities. The mechanism was evaluated with synthetic code and real web applications, and compared with other types of mechanisms that operate before the DBMS. The evaluation showed that SEPTIC performs better than the other mechanisms and detects injection attacks when real web applications are being monitored. Also, the place where it is inserted is the best to solve the semantic mismatch problem. The overhead of SEPTIC to the DBMS was evaluated, showing that it is low.

7.2 Future Work

The thesis described techniques for detection of vulnerabilities and protection of web applications. Future works can develop other tools and methodologies to improve the detection of vulnerabilities and protection of web applications, with the main goal of building secure software. We discuss some possible research directions.

Machine learning can be used to improve the dependability of computer systems. Differently of its application in this thesis, i.e., to detect vulnerabilities, it can be used in protection of web applications, detecting attacks of diverse vulnerability classes. An attack is constituted by malicious data exploring an attack vector, and a vector of attack is associated to a vulnerability class, meaning that its path contains properties that can be characterized and associated to the exploitation of a vulnerability class. Also, the malicious data contains some pattern(s), which are associated to the exploitation of a vulnerability class. Therefore, for each class of vulnerability, vector of attacks and malicious data, can be studied and characterized with properties and patterns, composing a data set with this knowledge. Then to apply machine learning classifiers to predict if the inserted data in web applications is malicious and constitutes an attack.

There are several static analysis tools that analyze the source code of web applications searching for input validation vulnerabilities, however, each one is tested and evaluated with a different set of applications, and for different scenarios. Creating a benchmark for these tools would allow comparing them. Each tool would be evaluated with a set of defined software quality metrics and real application scenarios, where each metric and scenario defines a goal to be reached by the tools.

Nowadays frameworks have been increasingly used in the development of web applications. These frameworks permit the combination of many programming or scripting languages and integrate them in one web application. Also, some of these frameworks contain an intermediate layer that can be used to sanitize and validate the entry points of the applications. However, the use of this intermediate layer does not invalidate the use of the best practices to write and build secure software. Furthermore, the programming and scripting languages continue to be "insecure", in the sense that a programmer could leave the applications with vulnerabilities. Therefore, new classes of vulnerabilities are emerging from these flaws and frameworks. To study these flaws and frameworks is a new challenge that can originate a new tool supporting different programming languages and their interconnections.

Bibliography

AHUJA, B., JANA, A., SWARNKAR, A. & HALDER, R. (2015). On preventing SQL injection attacks. *Advanced Computing and Systems for Security*, **395**, 49–64.

ALONSO, J.M., GUZMÁN, A., BELTRÁN, M. & BORDON, R. (2009). LDAP injection techniques. *Wireless Sensor Network*, **1**, 233–244.

ANTUNES, J., NEVES, N.F., CORREIA, M., VERISSIMO, P. & NEVES, R. (2010). Vulnerability removal with attack injection. *IEEE Transactions on Software Engineering*, **36**, 357–370.

ARISHOLM, E., BRIAND, L.C. & JOHANNESSEN, E.B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, **83**, 2–17.

BALL, T. (1999). The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference*, 216–234.

BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C. & VIGNA, G. (2008). Saner: composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 29th IEEE Symposium Security and Privacy*, 387–401.

BANABIC, R. & CANDEA, G. (2012). Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 281–294.

BANDHAKAVI, S., BISHT, P., MADHUSUDAN, P. & VENKATAKRISHNAN, V.N. (2007). CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 12–24.

BARRANTES, E.G., ACKLEY, D.H., PALMER, T.S., STEFANOVIC, D. & ZOVI, D.D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In

Proceedings of the 10th ACM Conference on Computer and Communications Security, 281–289.

BAUM, L.E. & PETRIE, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *The Annals of Mathematical Statistics*, **37**, 1554–1563.

BBC TECHNOLOGY (2014). Millions of websites hit by Drupal hack attack. Http://www.bbc.com/news/technology-29846539.

BERNERS-LEE, T., FIELDING, R. & MASINTER, L. (2005). Uniform resource identifier (URI): Generic syntax. IETF Request for Comments: RFC 3986.

BHOLE, A.T. & PATIL, A.I. (2014). Intrusion detection with hidden Markov model and Weka tool. *International Journal of Computer Applications (IJCA)*, **85**, 27–30.

BIGGAR, P. & GREGG, D. (2009). Static analysis of dynamic scripting languages. Draft: Monday 17th August, 2009 at 10:29.

BIGGAR, P., DE VRIES, E. & GREGG, D. (2009). A practical solution for scripting language compilers. In *Proceedings of the 24th ACM Symposium on Applied Computing*, 1916–1923.

BISHOP, M., BISHOP, M., DILGER, M. & DILGER, M. (1996). Checking for race conditions in file accesses. *Computing Systems*, **9**, 131–152.

BOYD, S.W. & KEROMYTIS, A.D. (2004). SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security Conference*, 292–302.

BRADSHAW, S. (2010a). Fuzzer automation with spike. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/.

BRADSHAW, S. (2010b). An introduction to fuzzing: Using fuzzers (spike) to find vulnerabilities. http://resources.infosecinstitute.com/intro-to-fuzzing/.

BRIAND, L.C., WÜST, J., DALY, J.W. & PORTER, D.V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, **51**, 245–273.

BUEHRER, G.T., WEIDE, B.W. & SIVILOTTI, P. (2005). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, 106–113.

BUGTRAQ (2015). http://www.securityfocus.com.

BUSH, W., PINCUS, J. & SIELAFF, D. (2000). A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, **30**, 775–802.

CADAR, C., DUNBAR, D. & ENGLER, D. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 209–224.

CECCHET, E., UDAYABHANU, V., WOOD, T. & SHENOY, P. (2011). Benchlab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*.

CHANDOLA, V., BANERJEE, A. & KUMAR, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, **41**, 15:1–15:58.

CHESS, B. & MCGRAW, G. (2004). Static Analysis for Security. *IEEE Security and Privacy*, **2**, 76–79.

CHESS, B. & WEST, J. (2007). *Secure programming with static analysis*. Addison-Wesley.

CHIPOUNOV, V., KUZNETSOV, V. & CANDEA, G. (2011). S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 265–278.

CLARKE, J. (2009). *SQL Injection Attacks and Defense*. Syngress.

COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P. & ZHANG, Q. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 63–78.

CVE (2015). http://cve.mitre.org.

DAHSE, J. & HOLZ, T. (2014). Simulation of built-in PHP features for precise static code analysis. In *Proceedings of the 21st Network and Distributed System Security Symposium*.

DAHSE, J. & HOLZ, T. (2015). Experience report: An empirical study of PHP security mechanism usage. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 60–70.

DAHSE, J., KREIN, N. & HOLZ, T. (2014). Code reuse attacks in PHP: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 42–53.

DB-ENGINES (2015). http://db-engines.com/en/ranking.

DE POEL, N.L. (2010). *Automated Security Review of PHP Web Applications with Static Code Analysis*. Master's thesis, State University of Groningen.

DEMILLO, R.A., LIPTON, R.J. & SAYWARD, F.G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, **11**, 34–41.

DEMŠAR, J. (2006). Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, **7**, 1–30.

DOUGLEN, A. (2007). SQL smuggling or, the attack that wasn't there. Tech. rep., COMSEC Consulting, Information Security.

DOUPÉ, A., CAVEDON, L., KRUEGEL, C. & VIGNA, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security '12, 26–26.

DOWD, M., MCDONALD, J. & SCHUH, J. (2006). *Art of Software Security Assessment*. Pearson Professional Education.

DUCHÈNE, F., RAWAT, S., RICHIER, J. & GROZ, R. (2013). Ligre: Reverse-engineering of control and data flow models for black-box XSS detection. In *Proceedings of the 20th Working Conference on Reverse Engineering*, 252–261.

DUCHÈNE, F., RAWAT, S., RICHIER, J. & GROZ, R. (2014). Kameleonfuzz: Evolutionary fuzzing for black-box XSS detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 37–48.

DURÃES, J. & MADEIRA, H. (2006). Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, **32**, 849–867.

ETOH, H. & YODA, K. (2002). ProPolice: Improved Stack-smashing Attack Detection. *Transactions of Information Processing Society of Japan*, **43**, 4034–4041.

EVANS, D. & LAROCHELLE, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, 42–51.

EVANS, D., GUTTAG, J., HORNING, J. & TAN, Y.M. (1994). Lclint: A tool for using specifications to check code. *SIGSOFT Software Engineering Notes*, **19**, 87–96.

EVRON, G. & RATHAUS, N. (2007). *Open Source Fuzzing Tools*. Elsevier Inc., 1st edn.

FONSECA, J. & VIEIRA, M. (2014). A practical experience on the impact of plugins in web security. In *Proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems*, 21–30.

FOSTER, J.S., FÄHNDRICH, M. & AIKEN, A. (1999). A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 192–203.

GÁLAN, E.C., ALCAIDE, A., ORFILA, A. & ALÍS, J.B. (2010). A multi-agent scanner to detect stored-xss vulnerabilities. In *Proceedings of the IEEE International Conference for Internet Technology and Secured Transactions*, 1–6.

GAMBAS (2015). http://gambas.sourceforge.net/.

GODEFROID, P., KLARLUND, N. & SEN, K. (2005). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 213–223.

GODEFROID, P., LEVIN, M.Y. & MOLNAR, D.A. (2008). Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*.

GODEFROID, P., LEVIN, M.Y. & MOLNAR, D. (2012). Sage: Whitebox fuzzing for security testing. *Communication ACM*, **55**, 40–44.

HALFOND, W. & ORSO, A. (2005). AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 174–183.

HALFOND, W., ORSO, A. & MANOLIOS, P. (2008). WASP: protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, **34**, 65–81.

HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M. & BOS, H. (2013). Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium*, 49–64.

HAN, J., KAMBER, M. & PEI, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edn.

HAND, D.J., SMYTH, P. & MANNILA, H. (2001). *Principles of Data Mining*. MIT Press.

HLADKÁ, B. & HOLUB, M. (2015). A gentle introduction to machine learning for natural language processing: How to start in 16 practical steps. *Language and Linguistics Compass*, **9**, 55–76.

HOWARD, G.M., GUTIERREZ, C.N., ARSHAD, F.A., BAGCHI, S. & QI, Y. (2014). pSigene: Webcrawling to generalize SQL injection signatures. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 45–56.

HOWARD, M. & LEBLANC, D. (2003). *Writing Secure Code. 2nd edition*. Microsoft Press.

HOWARD, M. & LEBLANC, D. (2007). *Writing Secure Code for Windows Vista*. Microsoft Press, 1st edn.

HUANG, J.C. (2009). *Software Error Detection through Testing and Analysis*. John Wiley and Sons, Inc.

HUANG, Y.W., HUANG, S.K., LIN, T.P. & TSAI, C.H. (2003). Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web*, 148–159.

HUANG, Y.W., YU, F., HANG, C., TSAI, C.H., LEE, D.T. & KUO, S.Y. (2004). Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, 40–52.

ICS-CERT (2015). Incident response/vulnerability coordination in 2014. ICS-CERT Monitor.

IMPERVA (2014). Anatomy of comment spam. hacker intelligence initiative.

IMPERVA (2015). Web application attack report #6.

JACKSON, D. & RINARD, M. (2000). Software analysis: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE'00, 133–145.

JIMENEZ, W., MAMMAR, A. & CAVALLI, A. (2009). Software vulnerabilities, prevention and detection methods: A review. In *Proceedings of the European Workshop on Security in Model Driven Architecture*, SEC-MDA'09, 6–13.

JOVANOVIC, N., KRUEGEL, C. & KIRDA, E. (2006). Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, 27–36.

JSOUP (2014). Jsoup. http://jsoup.org.

JURAFSKY, D. & MARTIN, J.H. (2008). *Speech and Language Processing*. Prentice Hall.

KAKSONEN, R. (2001). A functional method for assessing protocol implementation security. Tech. rep. 448, VTT.

KC, G.S., KEROMYTIS, A.D. & PREVELAKIS, V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 272–280.

KHOSRONEJAD, M., SHARIFIFAR, E., TORSHIZI, H.A. & JALALI, M. (2013). Developing a hybrid method of hidden Markov models and c5.0 as a intrusion detection system. *International Journal of Database Theory and Application*, **6**, 165–174.

KIEYZUN, A. ET AL. (2009). Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, 199–209.

KOSCHANY, M. (2013). Debian hardening. https://wiki.debian.org/ Hardening.

LANDI, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, **1**, 323–337.

LESSMANN, S., BAESENS, B., MUES, C. & PIETSCH, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, **34**, 485–496.

MARIADB (2015). Mariadb dbms. http://mariadb.org.

MASRI, W. & SLEIMAN, S. (2015). SQLPIL: SQL injection prevention by input labeling. *Security and Communication Networks*, **8**, 2545–2560.

MEASUREIT (2014). https://code.google.com/p/measureit/.

MEDEIROS, I. (2014). WAP website. http://awap.sourceforge.net/.

MEDEIROS, I. (2015). OWASP WAP - Web Application Protection. https://www.owasp.org/index.php/OWASP_WAP-Web_Application_Protection.

MERLO, E., LETARTE, D. & ANTONIOL, G. (2007). Automated Protection of PHP Applications Against SQL Injection Attacks. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 191–202.

MICHAEL, C. & LAVENHAR, S.R. (2006). Source Code Analysis Tools – Overview. https://buildsecurityin.us-cert.gov/articles/tools/source-code-analysis/source-code-analysis-tools—overview.

MILLER, B.P., FREDRIKSEN, L. & SO, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM*, **33**, 32–44.

MODELO-HOWARD, G., GUTIERREZAND, C., ARSHAD, F., BAGCHI, S. & QI, Y. (2014). Psigene: Webcrawling to generalize SQL injection signatures. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, 45–56.

MONGODB (2015). https://www.mongodb.org/.

NEUHAUS, S., ZIMMERMANN, T., HOLLER, C. & ZELLER, A. (2007). Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 529–540.

NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J. & EVANS, D. (2005). Automatically hardening web applications using precise tainting. *Security and Privacy in the Age of Ubiquitous Computing*, 295–307.

NIST (2016). NIST'S SAMATE. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.

NUNAN, A.E., SOUTO, E., DOS SANTOS, E.M. & FEITOSA, E. (2012). Automatic classification of cross-site scripting in web pages using document-based and url-based features. In *Proceedings of the IEEE Symposium on Computers and Communications*, 702–707.

NUNES, P., FONSECA, J. & VIEIRA, M. (2015). phpSAFE: A security analysis tool for OOP web application plugins. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

OPENSOURCETESTING.ORG (2015). Open Source Testing. http://www.opensourcetesting.org/security.php.

OSVDB (2015). http://osvdb.org.

OWASP (2013). Session fixation. https://www.owasp.org/index.php/Session_fixation.

OWASP (2014a). Owasp esapi. https://www.owasp.org/index.php/ESAPI.

OWASP (2014b). Testing for NoSQL injection. https://www.owasp.org/index.php/Testing_for_NoSQL_injection.

PACKET STORM (2015). https://packetstormsecurity.com.

PAPAGIANNIS, I., MIGLIAVACCA, M. & PIETZUCH, P. (2011). PHP Aspis: using partial taint tracking to protect against injection attacks. In *Proceedings of the 2nd USENIX Conference on Web Application Development*.

PARR, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf.

PARR, T. (2009). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf.

PERL, H., DECHAND, S., SMITH, M., ARP, D., YAMAGUCHI, F., RIECK, K., FAHL, S. & ACAR, Y. (2015). VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 426–437.

PHP ADDRESS BOOK (2015). http://php-addressbook.sourceforge.net.

PIETRASZEK, T. & BERGHE, C.V. (2005). Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, 124–145.

POSTGRESQL (2015). Postgresql dbms. http://www.postgresql.org/.

POWERS, D. (2015). Evaluation a monte carlo study. *CoRR*, **abs/1504.00854**, 843–844.

RABINER, L.R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, **77**, 257–286.

RASTHOFER, S., ARZT, S. & BODDEN, E. (2014). A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*.

RAY, D. & LIGATTI, J. (2012). Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 179–190.

REFBASE (2015). http://http://www.refbase.net.

RON, A., SHULMAN-PELEG, A. & BRONSHTEIN, E. (2015). No sql, no injection? examining nosql security. *CoRR*, **abs/1506.04082**.

SABELFELD, A. & MYERS, A.C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, **21**, 5–19.

SAMATE, N. (2014). Nist's samate reference dataset (srd). https://samate.nist.gov/SRD/.

SANDHU, R.S. (1993). Lattice-based access control models. *IEEE Computer*, **26**, 9–19.

SAXENA, P., HANNA, S., POOSANKAM, P. & SONG, D. (2010). FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the Network and Distributed System Security Symposium*.

SCAMBRAY, J., LUI, V. & SIMA, C. (2011). *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. Mc Graw Hill.

SCANDARIATO, R., WALDEN, J., HOVSEPYAN, A. & JOOSEN, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, **40**, 993–1006.

SEARCH SECURITY TECHTARGET (2015). Wordpress vulnerable to stored XSS. http://searchsecurity.techtarget.com/news/4500245137/WordPress-vulnerable-to-stored-XSS-researchers-find.

SELENIUM (2014). Selenium IDE. https://docs.seleniumhq.org.

SHANKAR, U., TALWAR, K., FOSTER, J.S. & WAGNER, D. (2001). Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*.

SHAR, L.K. & TAN, H.B.K. (2012a). Automated removal of cross site scripting vulnerabilities in web applications. *Information and Software Technology*, **54**, 467–478.

SHAR, L.K. & TAN, H.B.K. (2012b). Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*, 1293–1296.

SHAR, L.K. & TAN, H.B.K. (2012c). Predicting common web application vulnerabilities from input validation and sanitization code patterns. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 310–313.

SHAR, L.K., TAN, H.B.K. & BRIAND, L.C. (2013). Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 35th International Conference on Software Engineering*, 642–651.

SHIN, Y., MENEELY, A., WILLIAMS, L. & OSBORNE, J.A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, **37**, 772–787.

SMITH, N.A. (2011). *Linguistic Structure Prediction*. Graeme Hirst.

SON, S. & SHMATIKOV, V. (2011). SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.

SON, S., MCKINLEY, K.S. & SHMATIKOV, V. (2013). Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 1181–1192.

SOSKA, K. & CHRISTIN, N. (2014). Automatically detecting vulnerable websites before they turn malicious. In *Proceedings of the 23rd USENIX Security Symposium*, 625–640.

SPRING (2014a). Spring framework. http://spring.io/.

SPRING (2014b). Spring support. http://docs.spring.io/spring/docs/2.5.4/reference/aop.html.

SQLMAP (2014). sqlmap project. https://github.com/sqlmapproject/testenv/tree/master/mysql.

STUTTARD, D. & PINTO, M. (2007). *The Web Application Hacker's Handbook: Discovering and Exploiting Security*. Wiley Publishing, Inc.

SU, Z. & WASSERMANN, G. (2006). The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, 372–382.

SULTANA, A., HAMOU-LHADJ, A. & COUTURE, M. (2012). An improved hidden Markov model for anomaly detection using frequent common patterns. In *Proceedings of the IEEE International Conference on Communications*, 1113–1117.

SUTTON, M., GREENE, A. & AMINI, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 1st edn.

SYMANTEC (2013). Internet threat report. 2012 trends, volume 18.

SYMANTEC (2014). Internet threat report. 2013 trends, volume 19.

SYMANTEC (2015). Internet threat report. 2014 trends, volume 20.

T. BUDD ET AL. (1978). The design of a prototype mutation system for program testing. In *Proceedings of the AFIPS National Computer Conference*, 623–627.

TAN, L., ZHANG, X., MA, X., XIONG, W. & ZHOU, Y. (2008). AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium*, 379–394.

THE HACKER NEWS (2015). 600tb MongoDB database accidentally exposed on the internet. http://thehackernews.com/2015/07/MongoDB-Database-hacking-tool.html.

TRINH, M.T., CHU, D.H. & JAFFAR, J. (2014). S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 1232–1243.

TRUSTWAVE SPIDERLABS (2015). ModSecurity - Open Source Web Application Firewall. http://www.modsecurity.org.

VALEUR, F., MUTZ, D. & VIGNA, G. (2005). A learning-based approach to the detection of SQL attacks. In *Proceedings of the 2nd Detection of Intrusions and Malware, and Vulnerability Assessment*, 123–140.

VAN DE VEN, A. (2005). Limiting buffer overflows with execshield. Magazine 9, RedHat.

VIEGA, J., BLOCH, J., KOHNO, Y. & MCGRAW, G. (2000). Its4: a static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, 257–267.

VIEIRA, M., ANTUNES, N. & MADEIRA, H. (2009). Using web security scanners to detect vulnerabilities in web services. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks*.

VITERBI, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, **13**, 260–269.

WAGLE, P. & COWAN, C. (2003). Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, 243–255.

WAGNER, D., FOSTER, J.S., BREWER, E.A. & AIKEN, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, 3–17.

WALDEN, J., DOYLE, M., WELCH, G.A. & WHELAN, M. (2009). Security of open source web applications. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 545–553.

WANG, X., PAN, C., LIU, P. & ZHU, S. (2006). SigFree: A signature-free buffer overflow attack blocker. In *Proceedings of the 15th USENIX Security Symposium*, 225–240.

WANG, Y., LI, Z. & GUO, T. (2011). Program slicing stored XSS bugs in web application. In *Proceedings of the 5th IEEE International Conference on Theoretical Aspects of Software Engineering*, 191–194.

WASSERMANN, G. & SU, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 32–41.

WEBCHESS (2014). http://sourceforge.net/projects/webchess/.

WHITEHAT SECURITY (2015). Website security statistics report.

WILANDER, J. (2005). Modeling and visualizing security properties of code using dependence graphs. In *Proceedings of the 5th Conference on Software Engineering Research and Practice*, 65–74.

WILLIAMS, J. & WICHERS, D. (2010). OWASP Top 10 - the ten most critical web application security risks (2010). Tech. rep., OWASP Foundation.

WILLIAMS, J. & WICHERS, D. (2013). OWASP Top 10 2013 – the ten most critical web application security risks.

WITTEN, I.H., FRANK, E. & HALL, M.A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edn.

WORDPRESS (2015). https://wordpress.org/.

XU, W., BHATKAR, S. & SEKAR, R. (2005). Practical dynamic taint analysis for countering input validation attacks on web applications. Tech. Rep. SECLAB-05-04, Department of Computer Science, Stony Brook University.

YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H. & RIECK, K. (2013). Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 20th ACM SIGSAC Conference on Computer Communications Security*, 499–510.

YAMAGUCHI, F., GOLDE, N., ARP, D. & RIECK, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 590–604.

YAMAGUCHI, F., MAIER, A., GASCON, H. & RIECK, K. (2015). Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 797–812.

ZEROCMS (2014). Content management system built using PHP and MySQL. Http://www.aas9.in/zerocms/.

ZHENG, Y. & ZHANG, X. (2013). Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *Proceedings of the 2013 International Conference on Software Engineering*, 652–661.